

## Lecture2. Data Structures in Python for Data analysis

Good morning!  
Доброе утро!  
早上好!

**我们会成功**

**We will succeed !**

**У нас все получится [U nas vse poluchitsya] !**

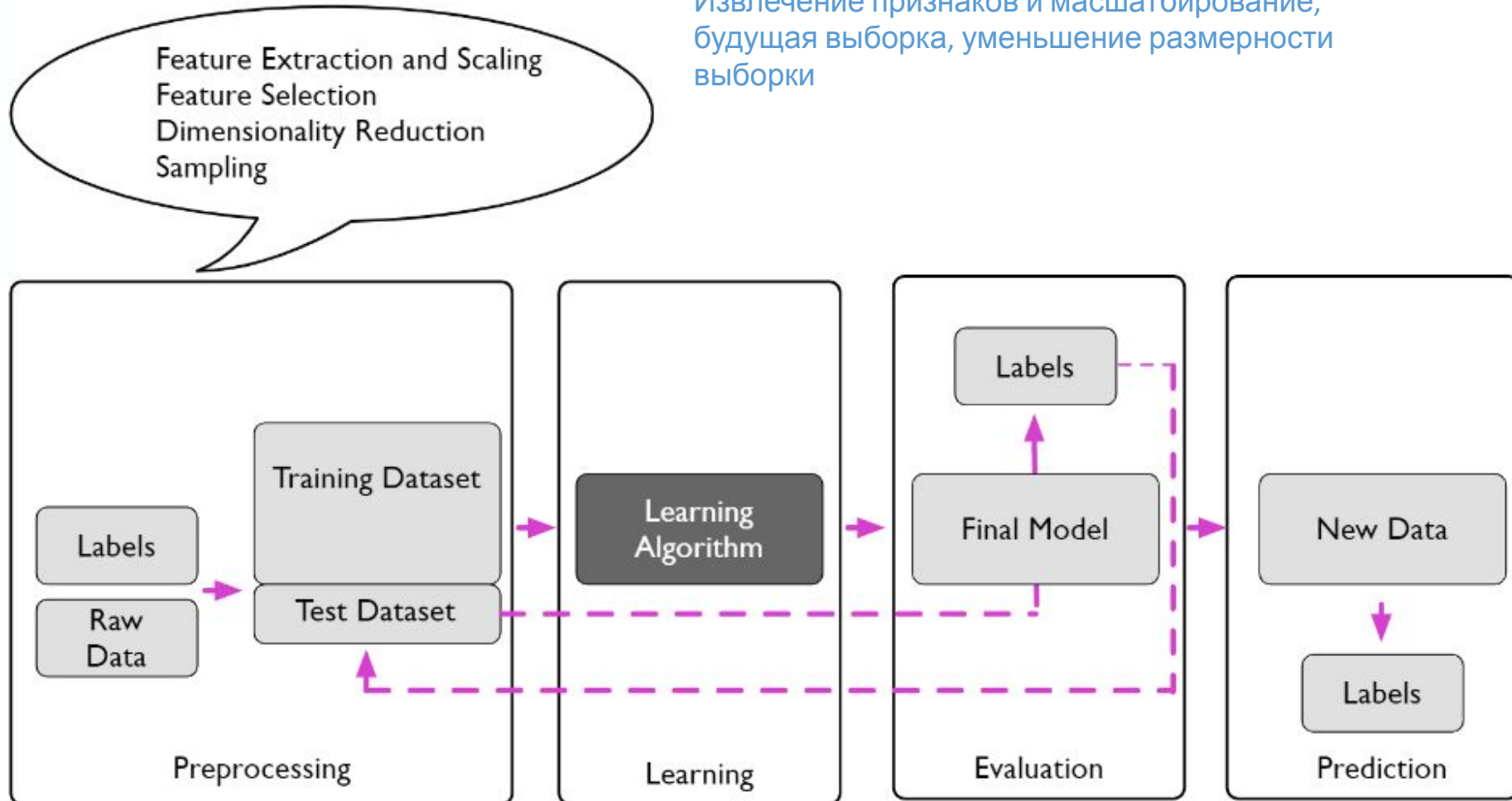
**Без булдырабыз!**



**Lecture3.**

**Data preprocessing and  
machine learning with  
Scikit-learn**

Извлечение признаков и масштабирование,  
будущая выборка, уменьшение размерности  
выборки



Model Selection  
Cross-Validation  
Performance Metrics  
Hyperparameter Optimization

# Training set and testing set

- Machine learning is about learning some properties of a data set and then testing those properties against another data set.
- A common practice in machine learning is to evaluate an algorithm by splitting a data set into two.
- We call one of those sets the **training set**, on which we learn some properties; we call the other set the **testing set**, on which we test the learned properties.

# Reading a Dataset

# The Iris Dataset



**Iris-Setosa**



**Iris-Versicolor**



**Iris-Virginica**

Fisher, R.A. "The use of multiple measurements in taxonomic problems" Annual Eugenics, 7, Part II, 179-188 (1936); also in "Contributions to Mathematical Statistics" (John Wiley, NY, 1950).

# Data Description :

Attribute Information:

1. sepal length in cm
2. sepal width in cm
3. petal length in cm
4. petal width in cm

Class:

- Iris Setosa
- Iris Versicolour
- Iris Virginica



- A basic table is a two-dimensional grid of data, in which the rows represent individual elements of the dataset, and the columns represent quantities related to each of these elements.
- For example, consider the [Iris dataset](#)

Here each row of the data refers to a single observed flower, and the number of rows is the total number of flowers in the dataset. In general, we will refer to the rows of the matrix as *samples*, and the number of rows as *n\_samples*.

The samples (i.e., rows) always refer to the individual objects described by the dataset. For example, the sample might be a flower, a person, a document, an image, a sound file, a video, an astronomical object, or anything else you can describe with a set of quantitative measurements.



	A	B	C	D	E
1	Sepal Length	Sepal Width	Petal Length	Petal Width	Class
2	5.1	3.5	1.4	0.2	Iris-setosa
3	4.9	3	1.4	0.2	Iris-setosa
4	4.7	3.2	1.3	0.2	Iris-setosa
5	4.6	3.1	1.5	0.2	Iris-setosa
6	5	3.6	1.4	0.2	Iris-setosa
7	5.4	3.9	1.7	0.4	Iris-setosa
8	4.6	3.4	1.4	0.3	Iris-setosa
9	5	3.4	1.5	0.2	Iris-setosa
10	4.4	2.9	1.4	0.2	Iris-setosa
11	4.9	3.1	1.5	0.1	Iris-setosa
...	...	...	...	...	...

# Target array

In dataset we also work with a *label* or *target* array, which by convention we will usually call  $y$ .

The target array is usually one dimensional, with length  $n\_samples$ . The target array may have continuous numerical values, or discrete classes/labels.

The distinguishing feature of the target array is that it is usually the quantity we want to *predict from the data*: in statistical terms, it is the **dependent** variable. For example, in the preceding data we may wish to construct a model that can predict the species of flower based on the other measurements; in this case, the species column would be considered the target array.

C	D	E
Petal Length	Petal Width	Class
1.4	0.2	Iris-setosa
1.4	0.2	Iris-setosa
1.3	0.2	Iris-setosa
1.5	0.2	Iris-setosa
1.4	0.2	Iris-setosa
1.7	0.4	Iris-setosa
1.4	0.3	Iris-setosa
1.5	0.2	Iris-setosa
1.4	0.2	Iris-setosa
1.5	0.1	Iris-setosa
...	...	...

- **Basic Data Analysis :**
- The dataset provided has 150 rows
- **Dependent Variables :** Sepal length, Sepal Width, Petal length, Petal Width
- **Independent/Target Variable :** Class
- Missing values : None

- The dataset is divided into Train and Test data with 80:20 split ratio where 80% data is training data where as 20% data is test data.

- Each training point belongs to one of  $N$  different classes.
- The goal is to construct a function which, given a new data point, will correctly predict the class to which the new point belongs.

Classification: samples belong to two or more classes and we want to learn from already labeled data how to predict the class of unlabeled data.

# What is scikit-learn?

- The scikit-learn library provides an implementation of a range of algorithms for Supervised Learning and Unsupervised Learning.

You can watch the Pandas and scikit-learn features documentation on this site.

- <https://pandas.pydata.org/pandas-docs/stable/>

<https://scikit-learn.org/stable/documentation.html>

# Preprocessing Data: missing data

- Real world data is filled with missing values.
- You will often need to rid your data of these missing values in order to train a model or do meaningful analysis.
- What follows are a few ways to impute (fill) missing values in Python, for both numeric and categorical data.



<b>idx</b>	<b>X1</b>	<b>X2</b>
0	NaN	0.875058
1	0.763618	0.772110
2	0.617353	0.643694
3	0.738487	0.542382
4	NaN	0.071095
5	0.716237	0.685460
6	0.730759	NaN
7	NaN	0.209360
8	0.710738	0.547641
9	0.546935	NaN

## Method 1: Mean or Median

- A common method of imputation with numeric features is to replace missing values with the mean of the feature's non-missing values. If the data have outliers, you may want to use the median instead. Either method is easy in Pandas:

```
df_mean_imputed = df.fillna(df.mean())  
df_median_imputed = df.fillna(df.median())
```

<b>idx</b>	<b>X1</b>	<b>X2</b>
0	0.689161	0.875058
1	0.763618	0.772110
2	0.617353	0.643694
3	0.738487	0.542382
4	0.689161	0.071095
5	0.716237	0.685460
6	0.730759	0.543350
7	0.689161	0.209360
8	0.710738	0.547641
9	0.546935	0.543350

# Imputation Method 2: Zero

- Depending on where your data are coming from, a missing value may be better represented by the number zero. Replacing missing values with zeros is accomplished similar to the above method; just replace the mean function with zero.

```
# replace missing values with the column mean
df_zero_imputed = df.fillna(0)
df_zero_imputed
```

<b>idx</b>	<b>X1</b>	<b>X2</b>
0	0.000000	0.875058
1	0.763618	0.772110
2	0.617353	0.643694
3	0.738487	0.542382
4	0.000000	0.071095
5	0.716237	0.685460
6	0.730759	0.000000
7	0.000000	0.209360
8	0.710738	0.547641
9	0.546935	0.000000

# Imputation for Categorical Data

- For categorical features, using mean, median, or zero-imputation doesn't make much sense. Here I'll create an example dataset with categorical features and show two imputation methods specific to this type of data.

idx	X1	X2
0	NaN	Green
1	Red	Green
2	Blue	Red
3	Red	Blue
4	NaN	Green
5	Red	Blue
6	Green	NaN
7	NaN	Red
8	Blue	Green
9	Red	NaN

# Imputation Method 1: Most Common Class

- One approach to imputing categorical features is to replace missing values with the most common class. You can do with by taking the index of the most common feature given in Pandas' `value_counts` function.

```
# for each column, get value counts in decreasing order and take the index (value) of most  
common class  
df_most_common_imputed = colors.apply(lambda x: x.fillna(x.value_counts().index[0]))
```



<b>idx</b>	<b>X1</b>	<b>X2</b>
0	Red	Green
1	Red	Green
2	Blue	Red
3	Red	Blue
4	Red	Green
5	Red	Blue
6	Green	Green
7	Red	Red
8	Blue	Green
9	Red	Green

# Imputation Method 2: “Unknown” Class

- Similar to how it's sometimes most appropriate to impute a missing numeric feature with zeros, sometimes a categorical feature's missing-ness itself is valuable information that should be explicitly encoded. If this is the case, most-common-class imputing would cause this information to be lost. Instead, just replace those values with a value like “Unknown” or “Missing.”
- `df_unknown_imputed = colors.fillna("Unknown")`

idx	X1	X2
0	Unknown	Green
1	Red	Green
2	Blue	Red
3	Red	Blue
4	Unknown	Green
5	Red	Blue
6	Green	Unknown
7	Unknown	Red
8	Blue	Green
9	Red	Unknown

# Column-Specific Imputation Rules

- You can combine any of the above methods by imputing specific columns rather than the entire dataframe. Returning to the numeric example, we can mean-impute X1 and median-impute X2 by specifying the column(s) to be imputed.

```
# replace missing values with the column mean
df['X1'] = df['X1'].fillna(df['X1'].mean())
df['X2'] = df['X2'].fillna(df['X2'].median())
df
```

# Preprocessing Data

## If data set are strings

- We saw in our initial exploration that most of the columns in our data set are strings, but the algorithms in scikit-learn understand only numeric data. Luckily, the scikit-learn library provides us with many methods for converting string data into numerical data. One such method is the `LabelEncoder()` method. We will use this method to convert the categorical labels in our data set like 'won' and 'loss' into numerical labels. To visualize what we are trying to achieve with the `LabelEncoder()` method let's consider the images below.

- The image below represents a dataframe that has one column named 'color' and three records 'Red', 'Green' and 'Blue'.

	Color
0	Red
1	Green
2	Blue

	Color
0	1
1	2
2	3

- Since the machine learning algorithms in scikit-learn understand only numeric inputs, we would like to convert the categorical labels like 'Red', 'Green' and 'Blue' into numeric labels. When we are done converting the categorical labels in the original dataframe, we would get something like this

```
#import the necessary module
from sklearn import preprocessing
# create the Labelencoder object
le = preprocessing.LabelEncoder()

#convert the categorical columns into numeric
encoded_value = le.fit_transform(["paris", "paris", "tokyo", "amsterdam"])

print(encoded_value)
```



```
[1 1 2 0]
```

he LabelEncoder() method assigns the numeric values to the classes in the order of the first letter of the classes from the original list: “(a)msterdam” gets an encoding of ‘0’ , “(p)aris gets an encoding of 1” and “(t)okyo” gets an encoding of 2.

# Training Set & Test Set

- A Machine Learning algorithm needs to be trained on a set of data to learn the relationships between different features and how these features affect the target variable.
- For this we need to divide the entire data set into two sets.
- One is the training set on which we are going to train our algorithm to build a model.
- The other is the testing set on which we will test our model to see how accurate its predictions are.



But before doing all this splitting, let's first separate our features and target variables.

```
data=df4[['DemAffl', 'DemAge', 'DemCluster', 'DemClusterGroup', 'DemGender', 'DemReg', 'DemTVReg', 'PromClass', 'PromSpend', 'PromTime']]
|
target = df4['TargetBuy']
data.head(2)
```

#import the necessary module

from sklearn.model\_selection import train\_test\_split

#split data set into train and test sets

data\_train, data\_test, target\_train, target\_test = train\_test\_split(data, target, test\_size = 0.30)

we used the train\_test\_split() method to divide the data into a training set (data\_train,target\_train) and a test set (data\_test,data\_train). The first argument of the train\_test\_split() method are the features that we separated out in the previous section, the second argument is the target('Opportunity Result'). The third argument 'test\_size' is the percentage of the data that we want to separate out as training data .

# Watch subtitled video

- <https://www.coursera.org/lecture/machine-learning/what-is-machine-learning-Ujm7v>





CHECK