- **Data dredging** (also known as **data snooping** or ***p*-hacking**)[1][a] is the misuse of data analysis to find patterns in data that can be presented as statistically significant, thus dramatically increasing and understating the risk of false positives

What is data dredging?

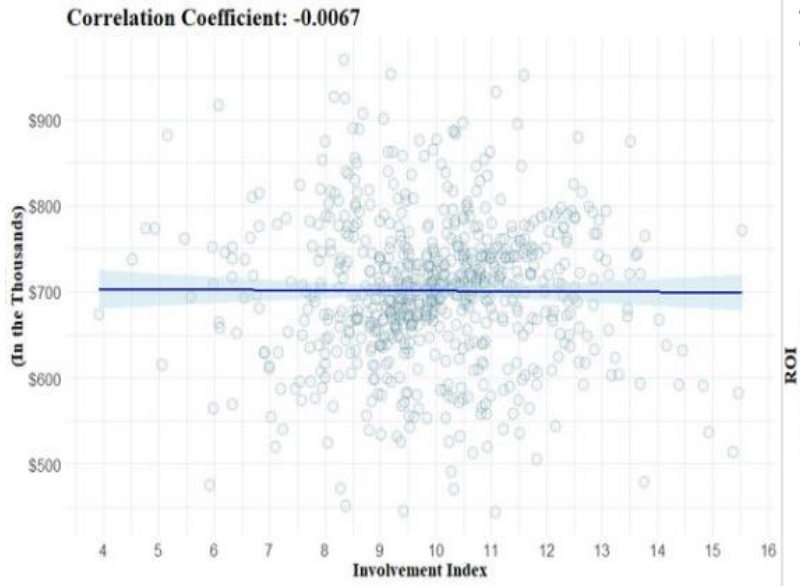
Imagine a marketing agency who, while trying to prove their business is worth it, asks an analyst (let's call her Sasha) to "prove" their return on investment.

The ask goes something like this:

> I'd like to be able to say that there is a significant correlation between our involvement index and our client's return on their investment so that we can attribute their success to us and not their other marketing affiliates.

Assume *involvement index* is standardized across all marketing vendors, and that this broad yet specific request will be proven by simply determining the correlation between the index and ROI.

Sasha begins by examining the relationship across all clients:

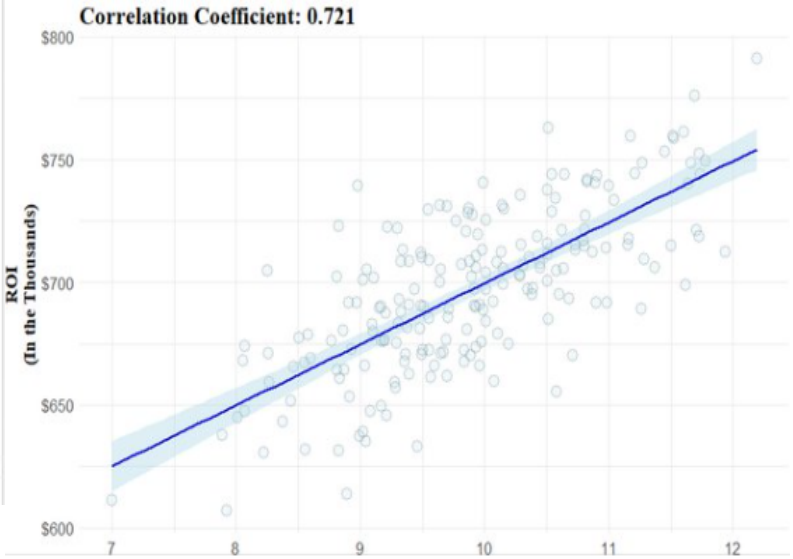


The data doesn't suggest there is likely to be a relationship, let alone a *positive* one.

In an **effort to please her leadership**, Sasha decides to start making some assumptions:

- removes clients with less than one year of business
- removes clients with more than 3.5 years of business
- removes clients with market-share in Washington D.C.
- removes clients in the automotive industry
- removes clients using the marketing vendor with the largest market share
- removes clients founded in 1979

After applying these assumptions, Sasha revisits the relationship between ROI and Involvement Index:



- Data pre-processing steps:

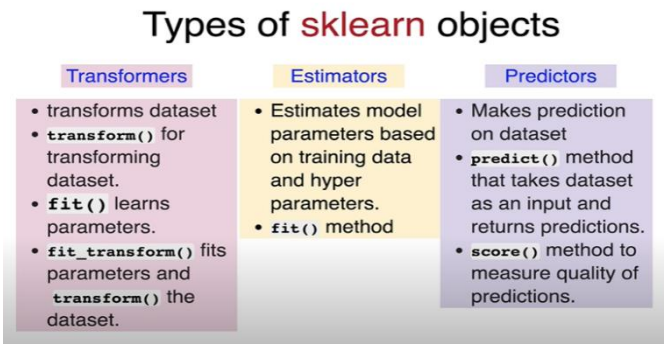
1. Separate features and labels.
2. Handling missing values and outliers.
3. Feature scaling to bring all features on the same scale.
4. Applying certain transformations like log, square root on the features.

- stratified sampling

- SimpleImputer: Replaces missing values using a descriptive statistic (e.g., mean, median, or most frequent) along each column, or using a constant value. Is an object and then used with fit, transform, fit_transform.

- fit() calculates the values of strategy eg mean, median or mean and variance for standardisation etc.

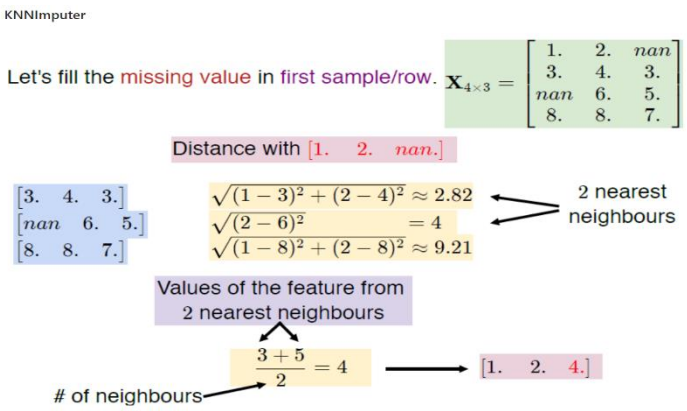
.transform() applies this calculated mean, median or standardisation formula using the calculated mean, var on the data.
.fit_transform() does the same job as above two simultaneously.



- Why do we use fit() and transform() or fit_transform() for training data and only transform() for test data ?

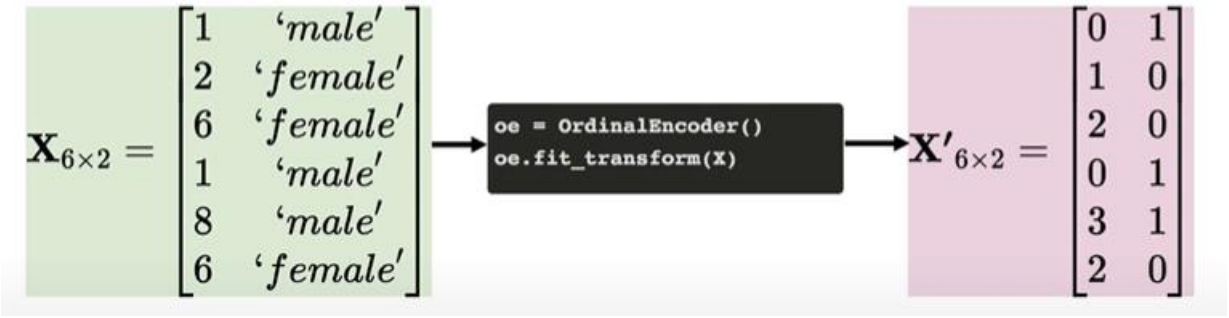
The fit method is calculating the mean and variance of each of the features present in our data. The transform method is transforming all the features using the respective mean and variance. Now, we want scaling to be applied to our test data too and at the same time do not want to be biased with our model. We want our test data to be a completely new and a surprise set for our model. The transform method helps us in this case. Using the transform method, we can use the same mean and variance as it is calculated from our training data to transform our test data. Thus, the parameters learned by our model using the training data will help us to transform our test data. Here is the simple logic behind it! If we will use the fit method on our test data too, we will compute a new mean and variance that is a new scale for each feature and will let our model learn about our test data too. Thus, what we want to keep as a surprise is no longer unknown to our model and we will not get a good estimate of how our model is performing on the test (unseen) data which is the ultimate goal of building a model using machine learning algorithm.

- KNNImputer: is used to fill the missing values like



- Convert categories to numbers using:

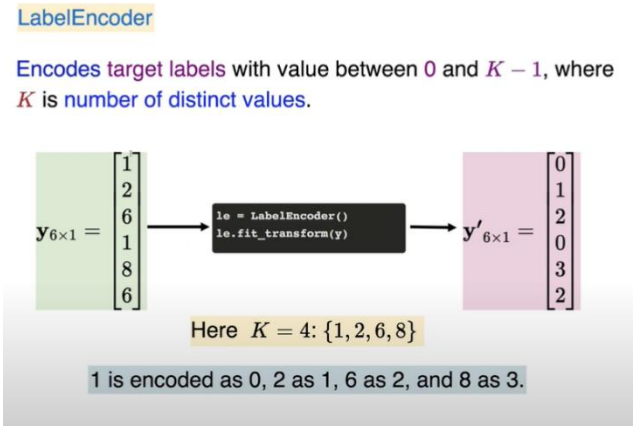
- OrdinalEncoder:



When the categorical feature is ordinal, we can use ordinal Encoding. Since the order among the categories is important, encoding should reflect the sequence. Can operate on multi-dim data.

- LabelEncoder:

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



- OneHotEncoder:

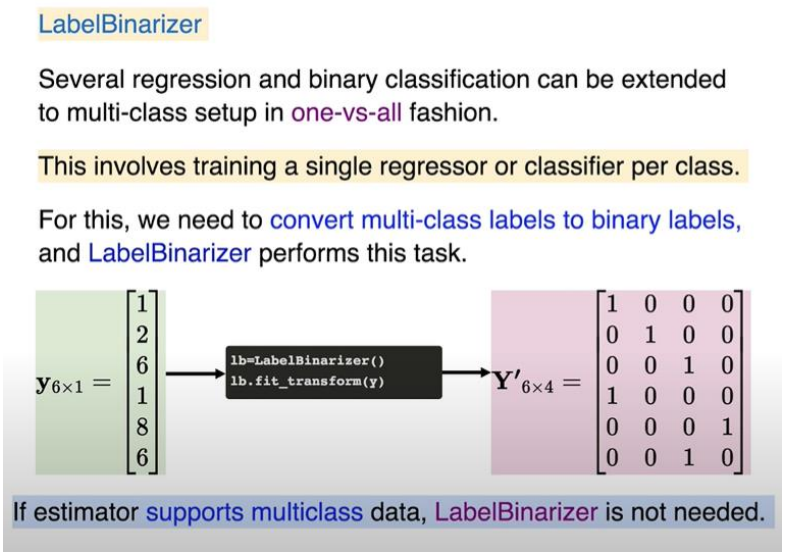
When the categorical features don't have any order (nominal), we use the One_Hot Encoding. In One_Hot Encoding each categorical column is split to many columns depending on the number of categories present in the column. Each column is mapped with 0 or 1s. Normally 1 represents an action that happened and 0 represents the action that did not happen.

Index	Animal
0	Dog
1	Cat
2	Sheep
3	Horse
4	Lion

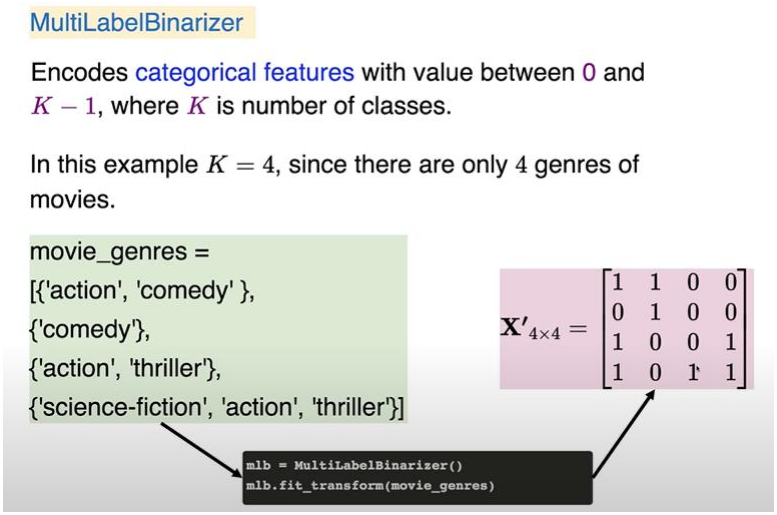
One-Hot code

Index	Dog	Cat	Sheep	Lion	Horse
0	1	0	0	0	0
1	0	1	0	0	0
2	0	0	1	0	0
3	0	0	0	0	1
4	0	0	0	1	0

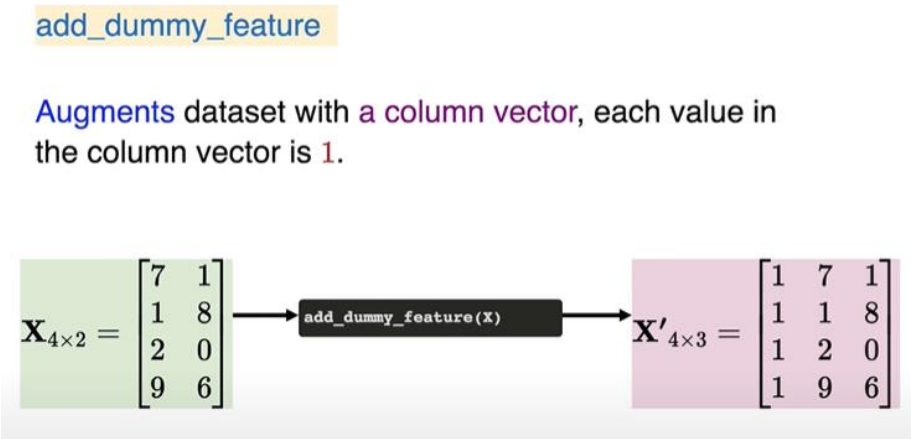
- LabelBinarizer:



- MultiLabelBinarizer:



- add_dummy_feature: we add dummy_feature for the bias term. y = wx1 +c, for the c term we add x0 which is our dummy_feature. Therefore, y = mx1 + cx0, where x0 corresponds to the bias.
  This is useful for adding a parameter for bias term in the model.



- Feature scaling: We want all the features on the same scale because:

Feature scaling enables faster convergence in iterative optimization algos like gradient descent and its variant.
The performance of ML algos like SVM, KNN and K-Means etc that compute Euclidean distance among input samples gets impacted if the features are not scaled.
Tress based ML algos do not require feature scaling.

- MaxAbsScaler/Normalization: The transformed feature vector will have all the values within the range [-1, 1].

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

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

- MinMaxScaler/Normalization: The transformed feature vector will have all the values within the range [0, 1].

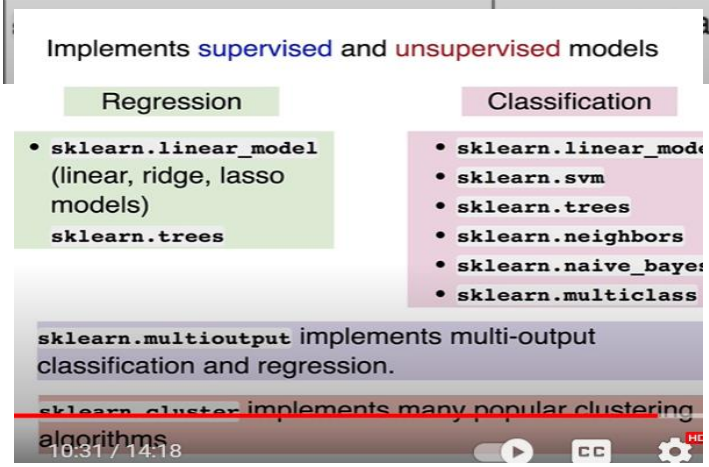
MinMaxScaler (Normalization) $$\mathbf{x}' = \frac{\mathbf{x} - \mathbf{x}.min}{\mathbf{x}.max - \mathbf{x}.min}$$

- StandardScaler/standardisation: The transformed feature vector will have a mean = 0 and SD = 1

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

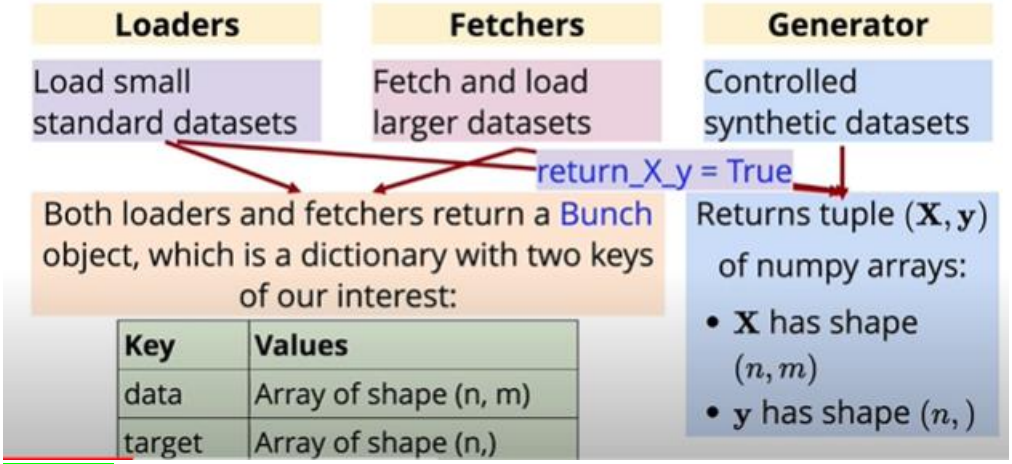
-

Module	Functionality
`sklearn.datasets`	Loading datasets - custom as well as popular reference dataset.
`sklearn.preprocessing`	Scaling, centering, normalization and binarization methods
`sklearn.impute`	Filling missing values
`sklearn.feature_selection`	Implements feature selection algorithms
	ature extraction from



sklearn.metrics implements different metrics for model evaluation

- sklearn.model_selection implements various model selection strategies like cross-validation, tuning hyperparameters, plotting learning curves.
- sklearn.model_inspection includes tools for model inspection



- LOADERS:

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

- FETCHERS:

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

- GENERATORS:

Regression: make_regression()
Classification: Single label: make_blobs() , make_classification()
Multi label: make_multilabel_classification()
Clustering: make_blobs()

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)

➢

➢ FEATURE EXTRACTION:

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

FeatureHasher:

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.

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

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

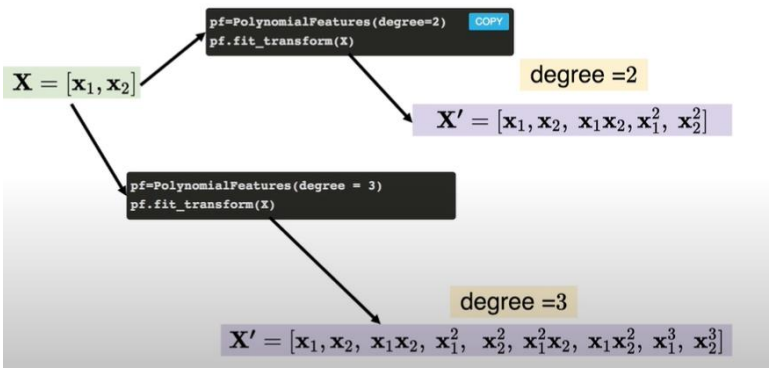
➢ FunctionTransformer: constructs transformed features by applying a user defined function.

Eg: FunctionTransformer(np.log2)

➢ PolynomialFeatures:

Polynomial transformation

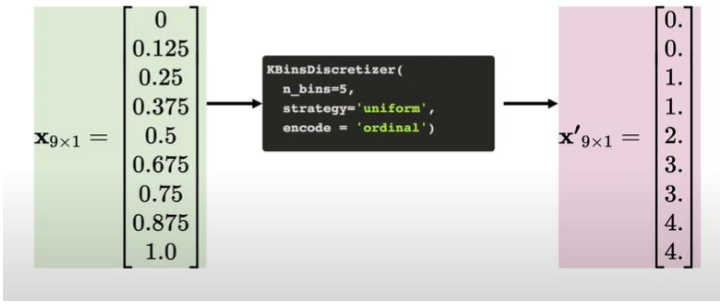
Generates a new feature matrix consisting of all polynomial combinations of the features with degree less than or equal to the specified degree.



➢ KBinsDiscretizer:

KBinsDiscretizer

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



➢ FEATURE SELECTION:

Filter based selection methods:

VarianceThreshold: Removes all features with variance below a certain threshold, as specified by the users.

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

Univariate feature selection method:

Selects feature based on univariate statistical tests.

There are three APIs for univariate feature selection:

SelectKBest: removes all but the K highest scoring features

SelectKBest

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

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

SelectPercentile: Removes all but a user-specified highest scoring percentage of features

SelectPercentile

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

COPY

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

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

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

COPY

- 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`.

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 an estimated 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:
- Filter based selection methods:

Univariate scoring function

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

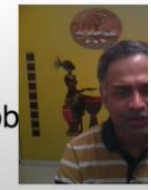
Mutual information (MI) | Chi-square | F-statistics

- 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 prob

chi2

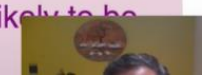


Mutual information (MI)

- 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.



- wrapper based selection methods:

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:

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'`.

SequentialFeatureSelection:

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.

➢ 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)
```

COPY

➢ TransformedTargetRegressor:

- `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)
```

COPY

This is how TransformedTargetRegressor works
: whenever we make prediction with transform target regressor we will
get output in the transform space which we translate back into the original space using the inverse function that is specified as an argument of the transform target regressor.

➢ Principal Component Analysis PCA:

- 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.

It is linear dimensionality reduction Technique.

➢ sklearn.pipeline :

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.

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 parameter of each step(estimators) 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)
```

pca is the <estimator> and n_components is the <parameterName>, therefore, pca__n_components. Similarly, for parameters of SimpleImputer and regressor estimators.

Performing GridSearch with the 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

FeatureUnion: combining transformers and pipeline. Below we have used pipeline for numeric features and column transformer for categorical features.

FeatureUnion has set up similar to that of pipeline.

- `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),])
```

COPY

➢ Data pre-processing flow:

- Data cleaning
  - Data Imputation
  - Feature scaling
- Feature transformation
  - Polynomial Features
  - Discretization
  - Handling categorical features
  - Custom Transformers
  - Composite Transformers
    - Apply transformation to diverse features
    - TargetTransformedRegressor
- Feature Selection
  - Filter based feature selection
  - Wrapper based feature selection
- Feature Extraction
  - PCA

➢ FunctionTransformer:

Enables conversion of an existing python function into a transformer to assist in data cleaning or processing

Useful when:

1. The dataset consists of heterogeneous data types (images, texts)
2. When different columns of dataset require different processing pipelines.
3. We need stateless transformations such as taking log of frequencies, custom scaling etc.

LINEAR REGRESSION:

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()
```

We use linear regression class for solving linear regression problem with normal equation method. And we use SGDRegressor class for solving linear regression problems with iterative optimization method.

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

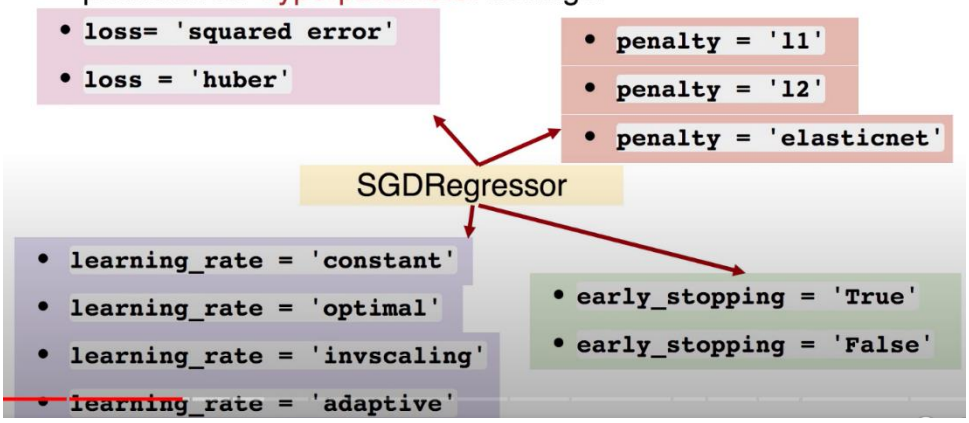
```
# Model training with feature matrix X_train and
# label vector or matrix y_train
linear_regressor.fit(X_train, y_train)
```

Works for both single and multi-output regression.

- Stochastic Gradient Descent:

SGD is used for training samples >10k

- Provides greater control on optimization process through provision for hyperparameter settings.



By default learning_rate is equal to "invscaling"

- Feature scaling is highly recommended for SGD

```
from sklearn.linear_model import SGDRegressor
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

sgd = Pipeline([
                ('feature_scaling', StandardScaler())),
                ('sgd_regressor', SGDRegressor())])

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.

What is the default setting?

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

Learning rate reduces after every iteration:

$$eta = eta0 / pow(t, power\_t)$$

eta0 is the initial learning rate value

- How to set “adaptive” learning rate?

  - 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? One epoch is one full pass over the training data. Default value is max_iter = 1000

```
from sklearn.linear_model import SGDRegressor
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?

Option 1: tol is the tolerance value for specific number of consecutive epochs, and the threshold value is represented by n_iter_no_change – these two arguments go together as a stopping criterion for the regressor. Otherwise, the regressor stops once the max_iter value is reached.

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

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

The SGDRegreesor 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`.

option 2:

Option #2 `early_stopping, validation_fraction`

```
from sklearn.linear_model import SGDRegressor
linear_regressor = SGDRegressor(loss='squared_error',
                                early_stopping=True
                                max_iter=500,
                                tol=1e-3,
                                validation_faction=0.2,
                                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 SGDRegreesor 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 averaged SGD?

option1: this updates the weight vector to average of weights from previous updates.

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

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

Option2: average = 10 (integer value) => starts averaging after seeing 10 samples

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

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

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

Sometimes when we do multiple runs of SGD regressor we want to use the value of the weight vector at the end of the previous SGD run as the initialization for the next SGD run. We can do that by setting warm_start parameter to true.

If the value of warm_start is false which is by default then every time we run the regressor it is initialized to new weights, and whatever work we have done in the previous run of regressor is ignored.

Make use of `warm_start = TRUE`

```
sgd_reg = SGDRegressor(max_iter=1, tol=-np.infty, warm_start=True,
                       penalty=None, learning_rate="constant", eta0=0.0005)

for epoch in range(1000):
    sgd_reg.fit(X_train, y_train)  # continues where it left off
    y_val_predict = sgd_reg.predict(X_val)
    val_error = mean_squared_error(y_val, y_val_predict)
```

Now, what we do is basically run this loop for 4000 times
and in every iteration, we call SGDRegressor fit object on training set. When we are calling the fit, we are actually using the value of the weight vector from the previous iteration, because we have set warm_start equal to true.

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_te
2 linear_regressor.predict(X_test)
```

Same code works for all regression estimators.

- Dummy regressor

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`

We instantiate a dummy regressor object by specifying the strategy for making the prediction. Here we specified the strategy as mean. That means we will be making predictions based on the mean value of the label. So, we learn this mean value of the label through the fit function on the training set. We will use this mean value later to predict for the test samples. And we calculate the score due to this prediction by using the test data. The score returns R square or coefficient of determination.

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

u = (predicted - actual) ^2, v = (actual – mean of predicted values) ^2

Using score method on linear regression object:

```
# Evaluation on the eval set with
# 1. feature matrix
# 2. label vector or matrix (single/multi-output)
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:
$u = (\mathbf{Xw} - \mathbf{y})^T(\mathbf{Xw} - \mathbf{y})$

Sum of squared error (actual and predicted label)

total sum of square

Sum of squared error (actual and mean predicted label)

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

the best possible score is 1

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).

All these metrics can also be used for multi-output datasets.

mean_absolute_error

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

mean_squarred_error

```
from sklearn.metrics import mean_squarred_error
eval_score = mean_squarred_error(y_test, y_predicted)
```

r2_score Same as output of **score**

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

These metrics can only be used for single output datasets

mean_squared_log_error

```
from sklearn.metrics import mean_squared_log_error
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

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

- Sensitive to relative error.

median_absolute_error

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

- Robust to outliers

- How to evaluate regression model on worst case error?

We use max_error function for that.

Use metrics `max_error`

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

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

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

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

- 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 to it.

Function	Scoring
metrics.mean_absolute_error	neg_mean_absolute_error
metrics.mean_squared_error	neg_mean_squared_error
metrics.mean_squared_error	neg_root_mean_squared_error
metrics.mean_squared_log_error	neg_mean_squared_log_error
metrics.median_absolute_error	neg_median_absolute_error

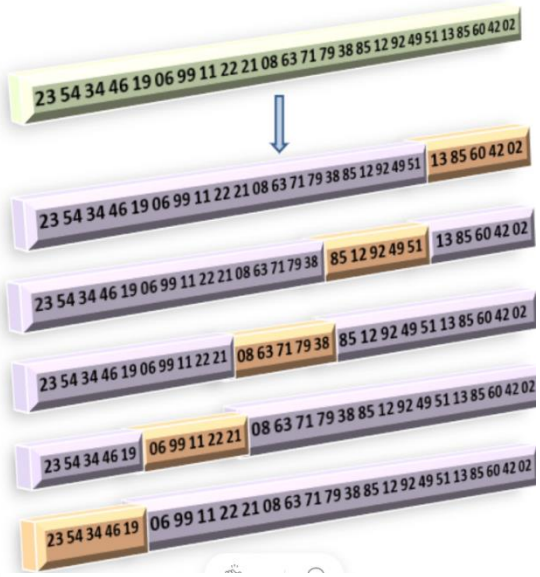
- Cross validation:

It performs robust evaluation of model performance by repeated splitting and by providing many training and test errors. This enables us to estimate variability in generalization performance of the model.

Cross validation iterators are as follows:

KFold:

K-1 folds for training and 1 for evaluation.



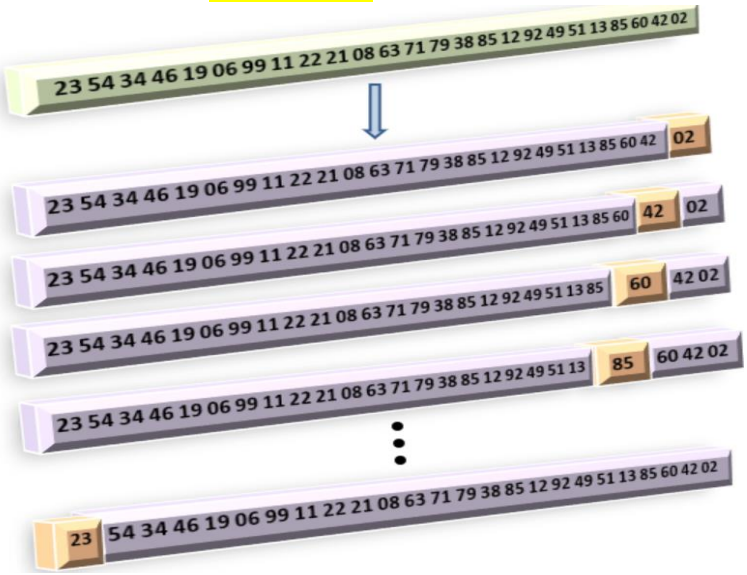

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import linear_regression

lin_reg = linear_regression()
score = cross_val_score(lin_reg, X, y, cv=5)
```

```
from sklearn.model_selection import cross_val_sc
from sklearn.model_selection import KFold
from sklearn.linear_model import linear_regression

lin_reg = linear_regression()
kfold_cv = KFold(n_splits=5, random_state=42)
score = cross_val_score(lin_reg, X, y, cv=kfold_cv)
```

LeaveOneOut:



The model is evaluated for every held-out observation. The result is then calculated by taking the mean of all the individual evaluations. This technique addresses the drawback of using small training sets, seen in the general validation set approach, as the model is fitted on almost all the training samples (n-1 observations)

```
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import LeaveOneOut
from sklearn.linear_model import linear_regression

lin_reg = linear_regression()
loocv = LeaveOneOut()
score = cross_val_score(lin_reg, X, y, cv=loocv)
```

which is same as

```
from sklearn.model_selection import cross_val_sc
from sklearn.model_selection import KFold
from sklearn.linear_model import linear_regression

lin_reg = linear_regression()
n = X.shape[0]
kfold_cv = KFold(n_splits=n)
score = cross_val_score(lin_reg, X, y, cv=kfold_cv)
```

ShuffleSplit/ random permutation:
In each iteration it shuffles order of data samples and then splits it into train and test.

```
from sklearn.linear_model import linear_regression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import ShuffleSplit

lin_reg = linear_regression()
shuffle_split = ShuffleSplit(n_splits=5, test_size=0.2, random_state=42)
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.

- How to specify performance measure (score) in cross_val_score?

```
from sklearn.linear_model import linear_regression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import ShuffleSplit

lin_reg = linear_regression()
shuffle_split = ShuffleSplit(n_splits=5, test_size=0.2, random_state=42)
score = cross_val_score(lin_reg, X, y, cv=shuffle_split,
                        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?

We use cross_validate for that instead of cross_val_score. Where data is x matrix and target is y label.

```
from sklearn.model_selection import cross_validate
from sklearn.model_selection import ShuffleSplit

cv = ShuffleSplit(n_splits=40, test_size=0.3, random_state=0)
cv_results = cross_validate(
    regressor, data, target, cv=cv, scoring="neg_mean_absolute_error")
```

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

fit_time

score_time

test_score

estimator (optional)

train_score (optional)

fit_time is the time taken to fit the regression model on the training set
score_time is time taken to get the score on evaluation set
test_score is the score obtained on the evaluation set. Here, it is neg_mean_absolute_error.

- Difference between cross_val_score and cross_validate:

- Cross_val_score runs single metric cross validation whilst cross_validate runs multi metric. This means that cross_val_score will only accept a single metric and return this for each fold, whilst cross_validate accepts a list of multiple metrics and will return all these for each fold.
- Cross_validate returns extra information not found in cross_val_score. In addition to the test scores, cross_validate also returns the fit times and score times.

- 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`

```
from sklearn.model_selection import cross_validate
from sklearn.model_selection import ShuffleSplit

cv = ShuffleSplit(n_splits=40, test_size=0.3,
                  random_state=0)
cv_results = cross_validate(
    regressor, data, target,
    cv=cv, scoring="neg_mean_absolute_error",
    return_train_score=True,
    return_estimator=True)
```

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

cross_validate allows us to specify multiple scoring metrics unlike cross_val_score

```
from sklearn.model_selection import cross_validate
from sklearn.model_selection import ShuffleSplit

cv = ShuffleSplit(n_splits=40, test_size=0.3,
                  random_state=0)
cv_results = cross_validate(
    regressor, data, target,
    cv=cv,
    scoring=["neg_mean_absolute_error", "neg_mean_squared_error"]
    return_train_score=True,
    return_estimator=True)
```

- Effect of no. 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.

```
from sklearn.model_selection import learning_curve

results = learning_curve(
    lin_reg, X_train, y_train, train_sizes=train_sizes, cv=cv,
    scoring="neg_mean_absolute_error")
train_size, train_scores, test_scores = results[:3]
# Convert the scores into errors
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.

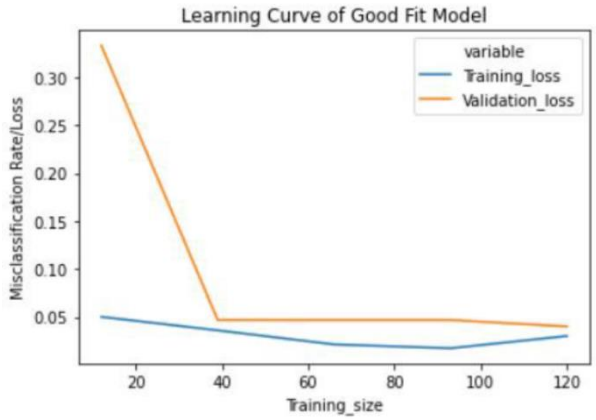
We can replace #features with any other tunable hyperparameter to do this diagnosis for setting that hyperparameter to the appropriate value.

- Generalization error = test error
- Empirical error = training error

- Learning curve:

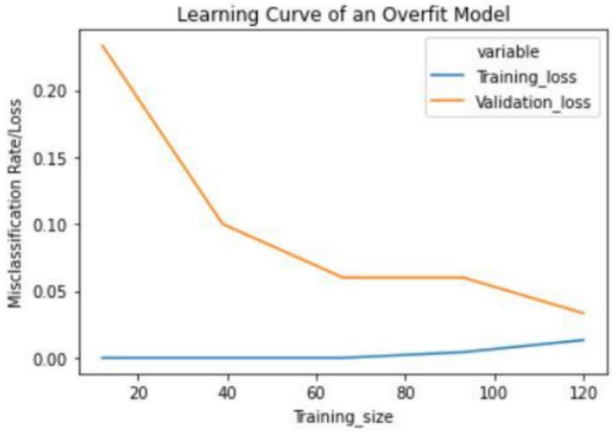
Typical features of the learning curve of a good fit model

1. Training loss and Validation loss are close to each other with validation loss being slightly greater than the training loss.
2. Initially decreasing training and validation loss and a flat training and validation loss after some point till the end.



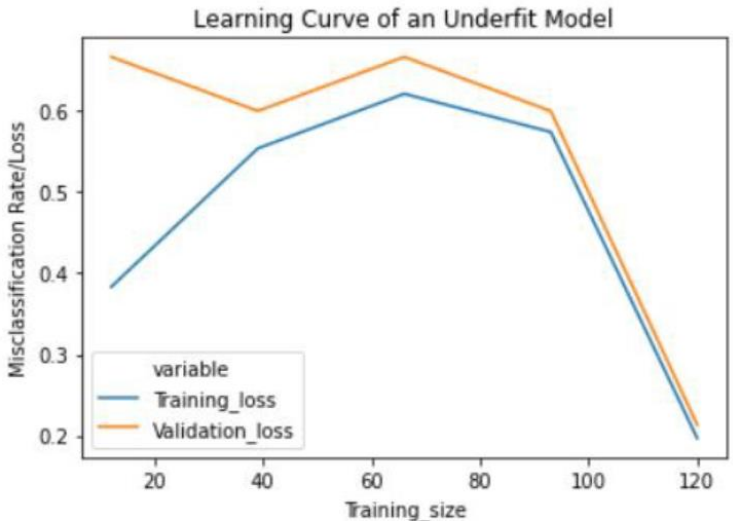
Typical features of the learning curve of an overfit model

1. Training loss and Validation loss are far away from each other.
2. Gradually decreasing validation loss (without flattening) upon adding training examples.
3. Very low training loss that is very slightly increasing upon adding training examples.



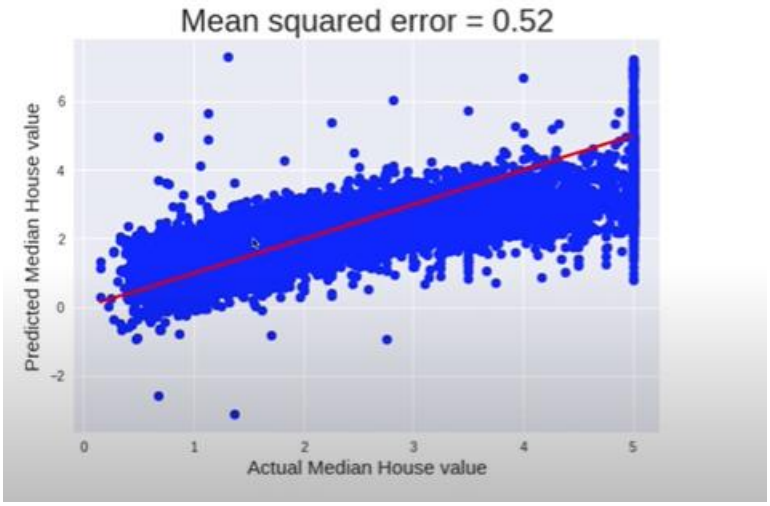
Typical features of the learning curve of an underfit model

1. Increasing training loss upon adding training examples.
2. Training loss and validation loss are close to each other at the end.
3. Sudden dip in the training loss and validation loss at the end (not always).




```
mse_cv = mean_squared_error(train_labels, cv_predictions)

plt.scatter(train_labels, cv_predictions, color='blue')
plt.plot(train_labels, train_labels, 'r-')
plt.title(f"Mean squared error = {mse_cv:.2f}", size=24)
plt.xlabel('Actual Median House value', size=15)
plt.ylabel('Predicted Median House value',size=15)
plt.show()
```



- POLYNOMIAL REGRESSION:

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.

- polynomial regression model with normal equation.

Set up polynomial regression model with normal equation

```
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

# Two steps:
# 1. Polynomial features of desired degree (here degree=2)
# 2. Linear regression
poly_model = Pipeline([
                ('polynomial_transform', PolynomialFeatures(degree=2))),
                ('linear_regression', LinearRegression())])

# Train with feature matrix X_train and label vector y_train
poly_model.fit(X_train, y_train)
```

- Polynomial regression model with iterative model

Set up polynomial regression model with SGD

```
from sklearn.linear_model import SGDRegressor
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

poly_model = Pipeline([
                ('polynomial_transform', PolynomialFeatures(degree=2))),
                ('sgd_regression', SGDRegressor())])
poly_model.fit(X_train, y_train)
```

- Only interaction features for polynomial regression:

```
from sklearn.preprocessing import PolynomialFeatures
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.

- Ridge regualrization(l2):

[Option #1]

Step 1: Instantiate object of Ridge estimator

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

```
from sklearn.linear_model import Ridge
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.

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

- How to search 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 SVDRegressor to search for best regularization.

- Grid search
- Randomized search

Similarly, can be used for lasso

- Lasso regualrization(l1):

[Option #1]

Step 1: Instantiate object of Lasso estimator

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

```
from sklearn.linear_model import Lasso
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.

- performing both lasso and ridge in polynomial regression.

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

```
from sklearn.linear_model import Lasso
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

poly_model = Pipeline([
                ('polynomial_transform', PolynomialFeatures(degree=2)),
                ('elasticnet', SGDRegressor(penalty='elasticnet',
                                            l1_ratio=0.3))])
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.

- How to recognize hyperparameters?

  Hyperparameters are the 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 the hyperparameters?

We select hyperparameters that results in the best cross-validation score.

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 hyperparameters tunning approaches implemented in sklearn

1. GridSearchCV: exhaustively considers all parameter combinations for specified values

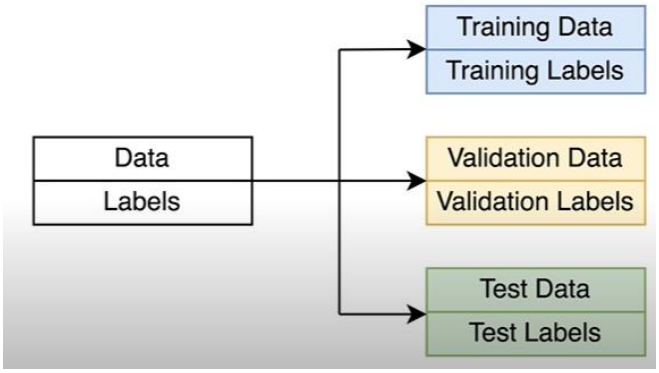
```
param_grid = [
  {'C': [1, 10, 100, 1000], 'kernel': ['linear']}
 ]
```

2. RandomizedSearchCV: samples a given number of candidate values from a parameter space with a specified distribution.

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

➢ Steps in hyperparameter tuning:

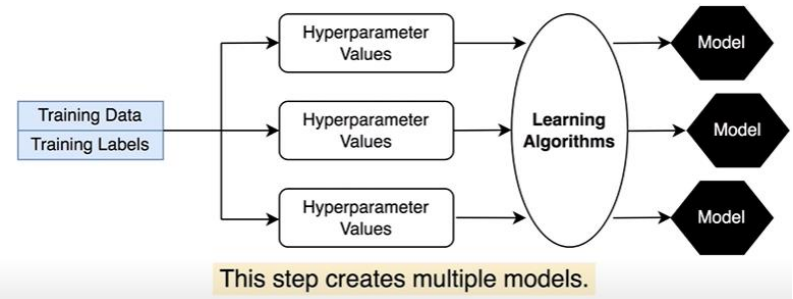
step 1



➢ Step 2

For each combination of hyperparameter values we learn a model with training set. The combination of hyperparameter values can be specified through Grid or Randomized CV.

This step creates multiple models.



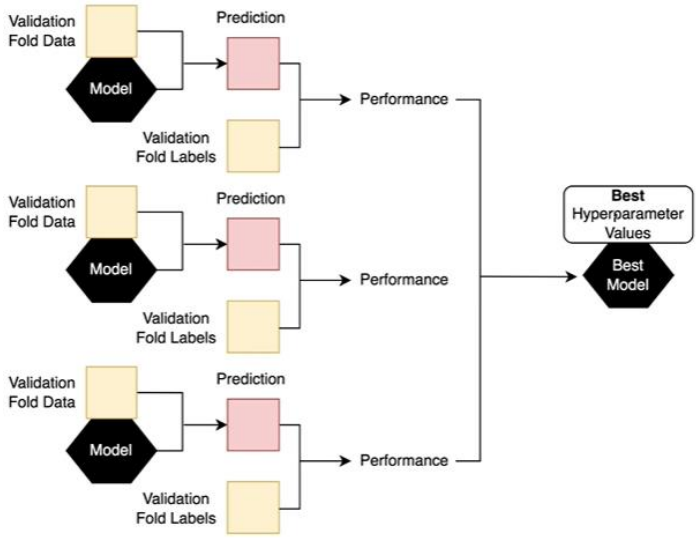
This step creates multiple models.

Tips

- This step can be 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.

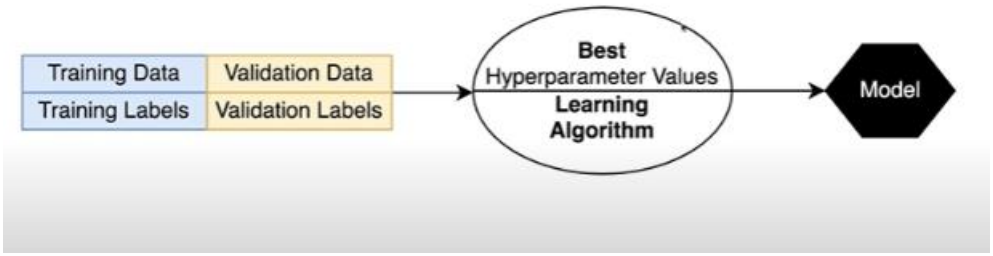
- Step 3

We use validation data on the model learnt in the previous step and get predictions. These predictions are then compared to the actual labels of validation set and performance score (e.g., MSE etc) is calculated. We repeat this step for each model and then the best model is selected. The selected model then has the best hyperparameter values



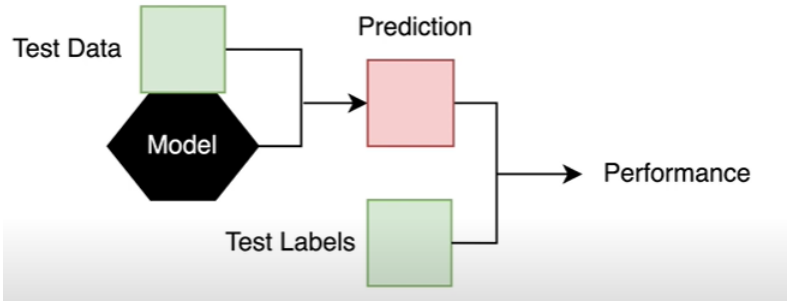
- Step 4:

Retrain model with the best hyper-parameter on the validation and train set combined



- Step 5

We use the model on the test set and evaluate the performance score on it



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

➢ Classification Functions in scikit:

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

Generic	Specific
• SGD classifier	• Logistic regression • Perceptron • Ridge classifier (for LSC) • K-nearest neighbours (KNNs) • Support vector machines (SVMs) • Naive Bayes
Uses gradient descent for opt	Specialized solvers for opt
Need to specify loss function	

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 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.

➢ Implementation of Least Square Classification with Ridge Classifier API:
Ridge classifier 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

the sample which gets predicted value as negative goes to -1 class and if predicted value is positive goes to +1 class

the norm of vector w square is the penalty regularized by alpha

Multiclass classification:
It is treated as multi-output regression
Predicted class corresponds with the output with highest value

Codes:

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

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

Default value of alpha = 0.1

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

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

- solvers are specific algos which solve optimization problems
  the choice of solvers depends on the classification problem setup such as size of the dataset, number of features and labels.

- solvers in Ridge classifier:

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.

For large scale data use “sparse_cg” solver
When both samples and features are large use “sag” or “saga” solvers
Note: fast convergence is only guaranteed on features with approximately the same scale.

Default choice for solver is auto, where auto chooses the solver automatically based on the type of data

Intercept estimation of Ridge Classifier:
If data is already centred, set fit_intercept as False, so that no intercept is used in calculations

RidgeClassifierCV implements RidgeClassifier with built-in cross validation.

- Perceptron classification with perceptron API

- 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.

Using Perceptron() is equivalent to using SGDClassifier() with the given arguments

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)

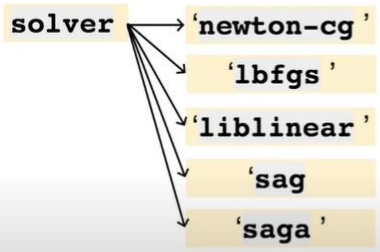
- Perceptron classifier can be trained in an iterative manner with partial_fit method
- Perceptron classifier can be initialized to the weights of the previous run by specifying warm_start = True in the constructor.

- Logistic regression classifier with LogisticRegression(logit regression/maximum entropy classifier(maxent)/log-linearclassifier) API:

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

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

➢ Solvers in LogisticRegression:



- 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 logstic reg uses lbfgs

➢ Penalty term:

- 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']

By default, logistic reg uses l2

➢ Controlling amount of regularization in Logistic reg:

- 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.

➢ class_weight parameter:

LogisticRegression classifier has a class_weight parameter in its constructor.

What purpose does it serve?

- Handles class imbalanace 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.

➢ cross validation in Logistic Regression:

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.

➢ 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

➢ loss parameter in SGDClassifier:

loss parameter

- 'hinge' - (soft-margin) linear Support Vector Machine
- 'log' - logistic regression
- 'modified_huber' - smoothed hinge loss brings tolerance to outliers as well as probability estimates
- 'squared_hinge' - like hinge but is quadratically penalized
- 'perceptron' - linear loss used by the perceptron algorithm
- 'squared_error', 'huber', 'epsilon_insensitive', or 'squared_epsilon_insensitive' - regression losses

by default sgd uses hinge loss

```
SGDClassifier(loss='log')
```
↓
```
LogisticRegression(solver='sgd')
```

these two examples show how different classifiers sometimes serve the same purpose. i.e., are equivalent to each other.

```
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.

➢ regularization in SGD

- *l*2 - adds a L2 penalty term
- *l*1 - adds a L1 penalty term

penalty

- 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:
```
SGD_classifier = SGDClassifier(penalty='l2')
```

➢ Alpha:

controls the amount of penalty and is a constant which is multiplied by regularization term

➢ 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

➢ Muti-learning classification:

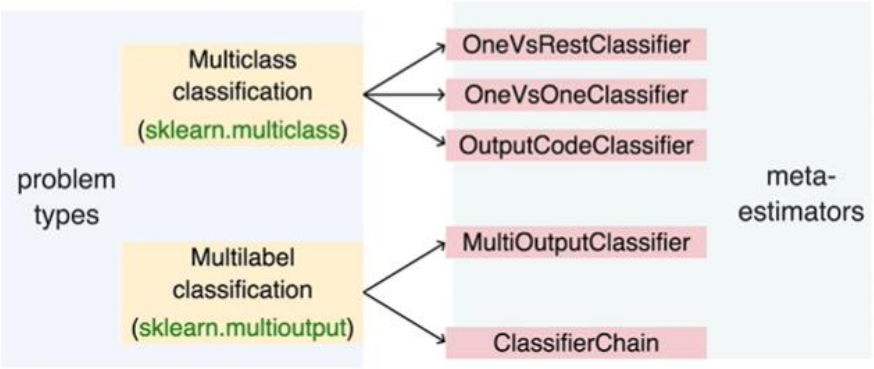
- Multiclass classification has exactly one output label and the total number of labels > 2.
- For more than one output, there are two types of classification models:

Multilabel	Multiclass multioutput
total #labels = 2	total #labels > 2

We will refer both these models as multi-label classification models, where # of output labels > 1.

➢ 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	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

Inherently multiclass | Multiclass as OVO | Multiclass as OVR | Multilabel

➢ Multiclass classification: have more than two classes, but has only one output label

- Use LabelBinarizer transformation to convert the class label to multi-class format.

```
from sklearn.preprocessing import LabelBinarizer
y = np.array(['apple', 'pear', 'apple', 'orange'])
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]]
```

➢ Type of target:

- Use type_of_target to determine the type of the label.

```
from sklearn.utils.multiclass import type_of_target
type_of_target(y)
```

target_type	y
'multiclass'	• contains more than two discrete values • not a sequence of sequences • 1d or a column vector
'multiclass-multioutput'	• 2d array that contains more than two discrete values • not a sequence of sequences • dimensions are of size > 1
'multilabel-indicator'	• label indicator matrix • an array of two dimensions with at least two columns, and at most 2 unique values.
'unknown'	• array-like but none of the above, such as a 3d array, • sequence of sequences, or an array of non-sequence objects.

multiclass

```
>>> type_of_target([1, 0, 2])
'multiclass'
>>> type_of_target([1.0, 0.0, 3.0])
'multiclass'
>>> type_of_target(['a', 'b', 'c'])
'multiclass'
```

multiclass-multioutput

```
>>> type_of_target(np.array([[1, 2], [3, 1]]))
'multiclass-multioutput'
```

multilabel-indicator

```
type_of_target(np.array([[0, 1], [1, 1]]))
'multilabel-indicator'
>>> type_of_target([[1, 2]])
'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

➢ Multiclass strategies:

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.

```
from sklearn.multiclass import OneVsRestClassifier
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.

```
from sklearn.multiclass import OneVsOneClassifier
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.

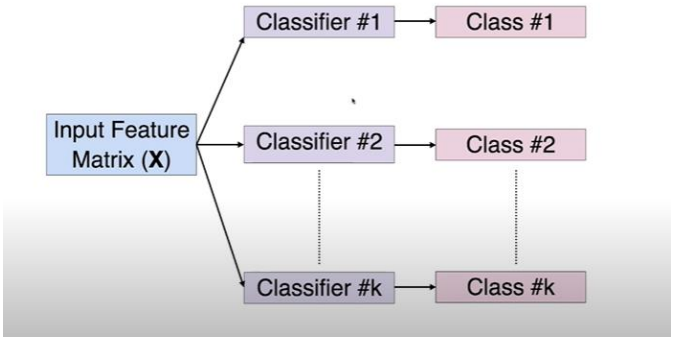
OneVsOne classifier processes subset of data at a time and is useful in cases where the classifier does not scale with the data.

What is the difference between OVR and OVA?

OneVsRestClassifier	OneVsOneClassifier
• Fits one classifier per class. • For each classifier, the class is fitted against all the other classes.	• Fits one classifier per pair of classes. • At prediction time, the class which received the most votes is selected.

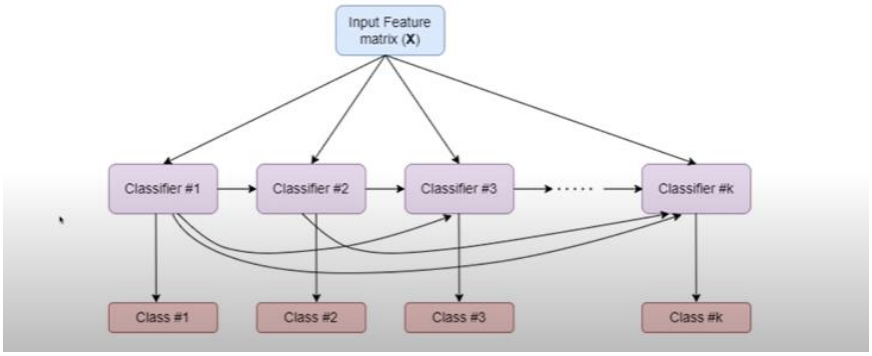
➢ Multi-label classifier:

- Strategy consists of fitting one classifier per target.



➢ Classifier chain:

- 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:

MultiOutputClassifier	ClassifierChain
• Able to estimate a series of target functions that are trained on a single predictor matrix to predict a series of responses.	• Capable of exploiting correlations among targets.
• Allows multiple target variable classifications.	• 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