Lecture 5
Data Preprocessing & Model Tuning

Phayung Meesad, Ph.D.
King Mongkut's University of Technology
North Bangkok (KMUTNB)
Bangkok Thailand

Outlines

- Data Preparation
- Pipelining
- Hyperparameter Tuning
- Model Optimizing

Data Preparation

- Sampling
- Normalization
- Binning/Discretization
- Missing
- Duplicates
- Outliers
- Dimensionality Reduction

Sampling

- Sampling is the main technique employed for data selection.
  - It is often used for both the preliminary investigation of the data and the final data analysis.
- Statisticians sample because obtaining the entire set of data of interest is too expensive or time consuming.
- Sampling is used in data mining because processing the entire set of data of interest is too expensive or time consuming.

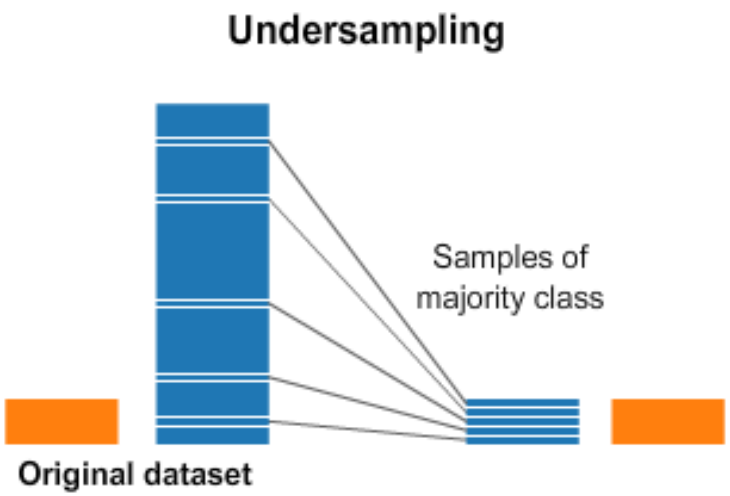
Sampling …

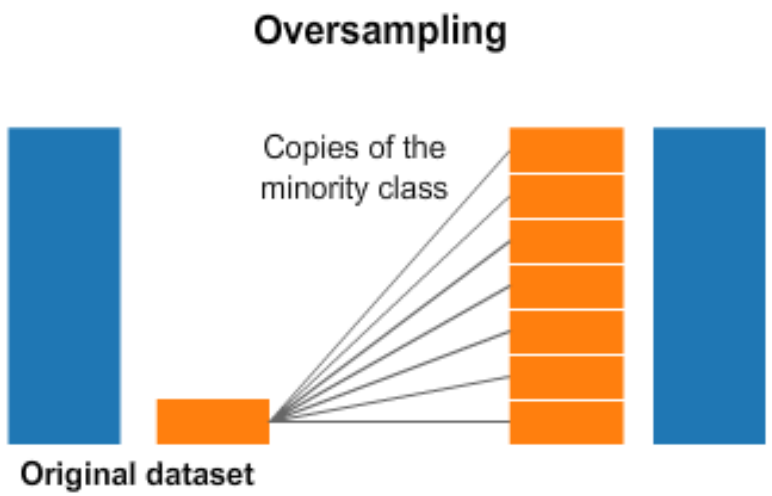
- The key principle for effective sampling is the following:
  - using a sample will work almost as well as using the entire data sets, if the sample is representative
  - A sample is representative if it has approximately the same property (of interest) as the original set of data

Types of Sampling

- Simple Random Sampling
  - equal probability of selecting any particular item
- Sampling without replacement
  - removed from the population
- Sampling with replacement (Bootstrap)
  - Objects are not removed from the population
- Stratified sampling
  - Split the data into several partitions; then draw random samples from each partition

Imbalanced Data





Data Balancing: 2 classes

```
import numpy as np
import pandas as pd
majority = [[5.1,3.5,1.4,0.2,0],
            [4.9,3.0,1.4,0.2,0],
            [4.7,3.2,1.3,0.2,0],
            [4.6,3.1,1.5,0.2,0],
            [5.0,3.6,1.4,0.2,0],
            [5.4,3.9,1.7,0.4,0],
            [4.6,3.4,1.4,0.3,0],
            [5.0,3.4,1.5,0.2,0],
            [4.4,2.9,1.4,0.2,0],
            [4.9,3.1,1.5,0.1,0]]
minority = [[7.0,3.2,4.7,1.4,1],
            [6.4,3.2,4.5,1.5,1],
            [6.9,3.1,4.9,1.5,1],
            [5.5,2.3,4.0,1.3,1],
            [6.5,2.8,4.6,1.5,1]]
names = ['A1','A2','A3','A4','labels']
```

```
majority_df = pd.DataFrame(majority,columns=names)
minority_df = pd.DataFrame(minority,columns=names)

def balance_data(majority_df, minority_df):
    fold=np.int32(np.floor(len(majority_df)/len(minority_df)))
    upsamples_minority=minority_df.copy()
    for i in range(0,fold):
        upsamples_minority=pd.concat([upsamples_minority, minority_df])

    train_data=pd.concat([majority_df,upsamples_minority])
    return train_data


data = balance_data(majority_df, minority_df)
data
```

SMOTE

- **S**ynthetic **M**inority **O**versampling **TE**chnique (SMOTE) was described by Nitesh Chawla, et al. (2002).
- SMOTE works by selecting examples that are close in the feature space, drawing a line between the examples in the feature space and drawing a new sample at a point along that line.
- SMOTE first selects a minority class instance a at random and finds its K nearest minority class neighbors.
- The synthetic instance is then created by choosing one of the K nearest neighbors b at random and connecting a and b to form a line segment in the feature space. The synthetic instances are generated as a convex combination of the two chosen instances a and b.

Data Normalization

- Range Normalization or MaxMin Scaler: to [new_minA, new_maxA]

$$v' = \frac{v - min_A}{max_A - min_A}(new\_max_A - new\_min_A) + new\_min_A$$

  - Ex. Let income range $12,000 to $98,000 normalized to [0.0, 1.0]. Then $73,600 is mapped to $\frac{73,600 - 12,000}{98,000 - 12,000}(1.0 - 0) + 0 = 0.716$

- Z-score normalization (μ: mean, σ: standard deviation) or StandardScaler:

$$v' = \frac{v - \mu_A}{\sigma_A}$$

  - Ex. Let μ = 54,000, σ = 16,000. Then $\frac{73,600 - 54,000}{16,000} = 1.225$

- Normalization by decimal scaling

$$v' = \frac{v}{10^j} \quad \text{where } j \text{ is the smallest integer such that } \text{Max}(|v'|) < 1$$

MinMaxScaler

- sklearn.preprocessing.MinMaxScaler(feature_range=(0, 1), copy=True)

Parameters:	**feature_range** : *tuple (min, max), default=(0, 1)*Desired range of transformed data. **copy** : *boolean, optional, default True*Set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array).

MinMaxScaler

Attributes:	**min_** : *ndarray, shape (n_features,)* Per feature adjustment for minimum. Equivalent to min - X.min(axis=0) * self.scale_ **scale_** : *ndarray, shape (n_features,)* Per feature relative scaling of the data. Equivalent to(max - min) / (X.max(axis=0) - X.min(axis=0)) *New in version 0.17: scale_* attribute. **data_min_** : *ndarray, shape (n_features,)* Per feature minimum seen in the data *New in version 0.17: data_min_* **data_max_** : *ndarray, shape (n_features,)* Per feature maximum seen in the data *New in version 0.17: data_max_* **data_range_** : *ndarray, shape (n_features,)* Per feature range (data_max_ - data_min_) seen in the data *New in version 0.17: data_range_*

MinMaxScaler

- fit(self, X[, y]) Compute the minimum and maximum to be used for later scaling.
- fit_transform(self, X[, y]) Fit to data, then transform it.
- get_params(self[, deep]) Get parameters for this estimator.
- inverse_transform(self, X) Undo the scaling of X according to feature_range.
- partial_fit(self, X[, y]) Online computation of min and max on X for later scaling.
- set_params(self, **params) Set the parameters of this estimator.
- transform(self, X) Scaling features of X according to feature_range.

```
from sklearn.preprocessing import MinMaxScaler
data = [[-1, 2], [-0.5, 6], [0, 10], [1, 18]]
scaler = MinMaxScaler()
print(scaler.fit(data))
print(scaler.data_max_)
print(scaler.transform(data))
print(scaler.transform([[2, 2]]))
```

StandardScaler

- sklearn.preprocessing.StandardScaler(copy=True, with_mean=True, with_std=True)

Parameters:	**copy** : *boolean, optional, default True* If False, try to avoid a copy and do inplace scaling instead. This is not guaranteed to always work inplace; e.g. if the data is not a NumPy array or scipy.sparse CSR matrix, a copy may still be returned. **with_mean** : *boolean, True by default* If True, center the data before scaling. This does not work (and will raise an exception) when attempted on sparse matrices, because centering them entails building a dense matrix which in common use cases is likely to be too large to fit in memory. **with_std** : *boolean, True by default* If True, scale the data to unit variance (or equivalently, unit standard deviation).

StandardScaler

Attributes:	**scale_** : *ndarray or None, shape (n_features,)* Per feature relative scaling of the data. This is calculated using np.sqrt(var_). Equal to Nonewhen with_std=False. *New in version 0.17: scale_* **mean_** : *ndarray or None, shape (n_features,)* The mean value for each feature in the training set. Equal to None when with_mean=False. **var_** : *ndarray or None, shape (n_features,)* The variance for each feature in the training set. Used to compute scale_. Equal to Nonewhen with_std=False. **n_samples_seen_** : *int or array, shape (n_features,)* The number of samples processed by the estimator for each feature. If there are not missing samples, the n_samples_seen will be an integer, otherwise it will be an array. Will be reset on new calls to fit, but increments across partial_fit calls.

Method of StandardScaler

- fit(self, X[, y]) Compute the mean and std to be used for later scaling.
- fit_transform(self, X[, y]) Fit to data, then transform it.
- get_params(self[, deep])Get parameters for this estimator.
- inverse_transform(self, X[, copy]) Scale back the data to the original representation
- partial_fit(self, X[, y]) Online computation of mean and std on X for later scaling.
- set_params(self, **params) Set the parameters of this estimator.
- transform(self, X[, copy]) Perform standardization by centering and scaling

```
from sklearn.preprocessing import StandardScaler
data = [[0, 0], [0, 0], [1, 1], [1, 1]]
scaler = StandardScaler()
print(scaler.fit(data))
print(scaler.mean_)
print(scaler.transform(data))
print(scaler.transform([[2, 2]]))
```

Discretization

- By Size: Each bin contains a user-defined number of examples.
- By Binning: Bins of equal range are automatically generated, the number of the values in different bins may vary.
- By Frequency: Bins of equal frequency are automatically generated, the range of different bins may vary.
- By Entropy: The boundaries of the bins are chosen so that the entropy is minimized in the induced partitions.

KBinsDiscretizer

- sklearn.preprocessing.KBinsDiscretizer(n_bins=5, encode='onehot', strategy='quantile')

Parameters:	**n_bins** : *int or array-like, shape (n_features,) (default=5)* The number of bins to produce. Raises ValueError if n_bins < 2. **encode** : *{'onehot', 'onehot-dense', 'ordinal'}, (default='onehot')* Method used to encode the transformed result. Onehot Encode the transformed result with one-hot encoding and return a sparse matrix. Ignored features are always stacked to the right. onehot-dense Encode the transformed result with one-hot encoding and return a dense array. Ignored features are always stacked to the right. Ordinal Return the bin identifier encoded as an integer value. **strategy** : *{'uniform', 'quantile', 'kmeans'}, (default='quantile')* Strategy used to define the widths of the bins. Uniform All bins in each feature have identical widths. Quantile All bins in each feature have the same number of points. Kmeans Values in each bin have the same nearest center of a 1D k-means cluster.

KBinsDiscretizer

Methods

fit(self, X[, y])	Fits the estimator.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, Xt)	Transforms discretized data back to original feature space.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Discretizes the data.

Missing Values

- Reasons for missing values
  - Information is not collected
    (e.g., people decline to give their age and weight)
  - Attributes may not be applicable to all cases
    (e.g., annual income is not applicable to children)

- Handling missing values
  - Remove Unused Values
  - Replace Missing Values
  - Impute Missing Values

SimpleImpute

- sklearn.impute.SimpleImputer(missing_values=nan, strategy='mean', fill_value=None, verbose=0, copy=True, add_indicator=False)

Parameters:

missing_values : *number, string, np.nan (default) or None*
The placeholder for the missing values. All occurrences of missing_values will be imputed.
strategy : *string, optional (default="mean")*The imputation strategy.
•If "mean", then replace missing values using the mean along each column. Can only be used with numeric data.
•If "median", then replace missing values using the median along each column. Can only be used with numeric data.
•If "most_frequent", then replace missing using the most frequent value along each column. Can be used with strings or numeric data.
•If "constant", then replace missing values with fill_value. Can be used with strings or numeric data.
New in version 0.20: strategy="constant" for fixed value imputation.
fill_value : *string or numerical value, optional (default=None)*When strategy == "constant", fill_value is used to replace all occurrences of missing_values. If left to the default, fill_value will be 0 when imputing numerical data and "missing_value" for strings or object data types.
verbose : *integer, optional (default=0)* Controls the verbosity of the imputer.
copy : *boolean, optional (default=True)*If True, a copy of X will be created. If False, imputation will be done in-place whenever possible. Note that, in the following cases, a new copy will always be made, even if copy=False:
•If X is not an array of floating values;
•If X is encoded as a CSR matrix;
•If add_indicator=True.
add_indicator : *boolean, optional (default=False)*
If True, a MissingIndicator transform will stack onto output of the imputer's transform. This allows a predictive estimator to account for missingness despite imputation. If a feature has no missing values at fit/train time, the feature won't appear on the missing indicator even if there are missing values at transform/test time.

SimpleImpute

Methods

fit(self, X[, y])	Fit the imputer on X.
fit_transform(self, X[, y])	Fit to data, then transform it.
get_params(self[, deep])	Get parameters for this estimator.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Impute all missing values in X.

SimpleImpute

```
import numpy as np
from sklearn.impute import SimpleImputer
imp = SimpleImputer(missing_values=np.nan,
strategy='mean')
imp.fit([[1, 2], [np.nan, 3], [7, 6]])

X = [[np.nan, 2], [6, np.nan], [7, 6]]
print(imp.transform(X))
```

KNNImputer

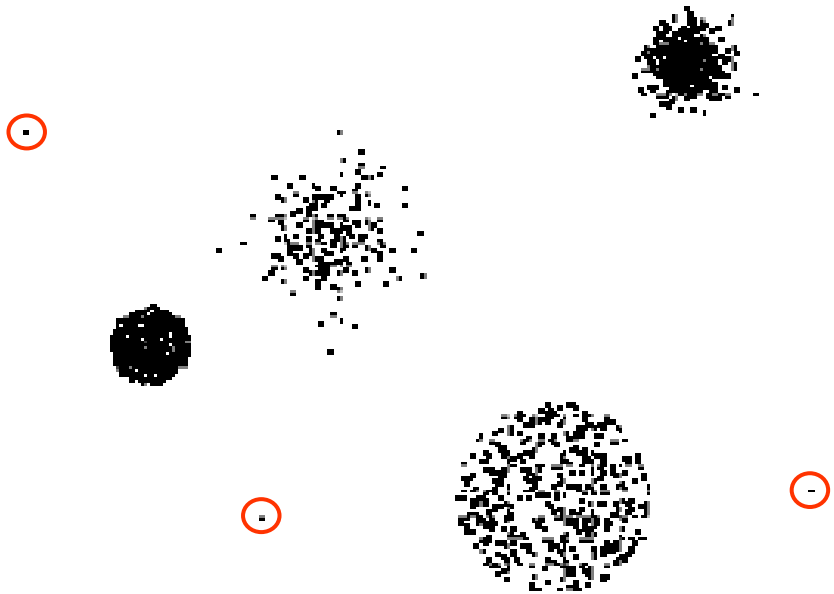
```
import numpy as np
from sklearn.impute import KNNImputer
nan = np.nan
X = [[1, 2, nan], [3, 4, 3],
     [nan, 6, 5], [8, 8, 7]]
imputer = KNNImputer(n_neighbors=2,
                     weights="uniform")
imputer.fit_transform(X)
```

Duplicate Data

- Data set may include data objects that are duplicates, or almost duplicates of one another
  - Major issue when merging data from heterogeneous sources
- Examples:
  - Same person with multiple email addresses
- Data cleaning
  - Process of dealing with duplicate data issues

Outliers

- Outliers are data objects with characteristics that are considerably different than most of the other data objects in the data set



Dimensionality Reduction

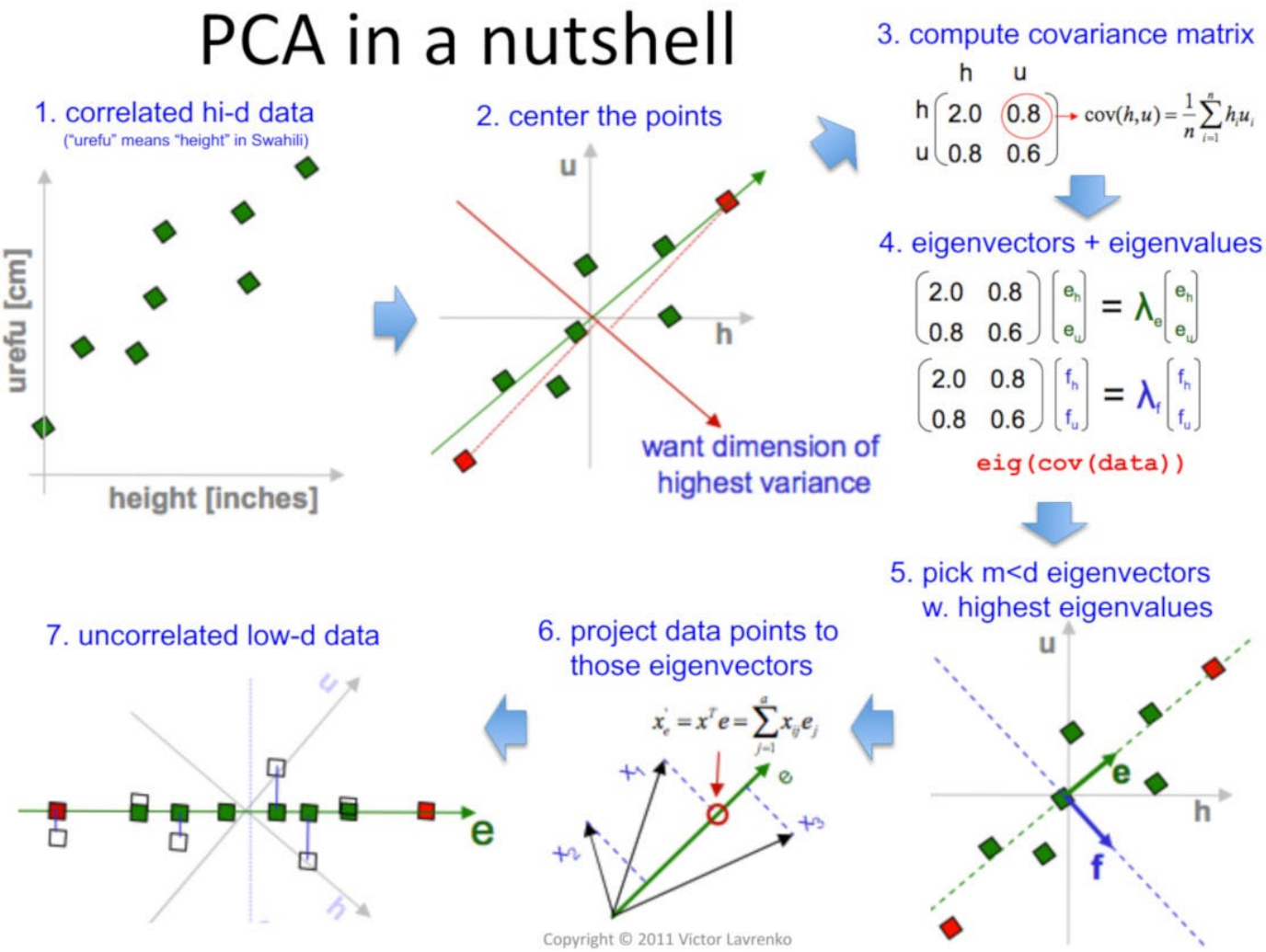
- Purpose:
  - Avoid curse of dimensionality
  - Reduce amount of time and memory
  - Allow data to be more easily visualized
  - Eliminate irrelevant features or reduce noise

- Techniques
  - Principle Component Analysis (PCA)
  - Independent Component Analysis (ICA)
  - Singular Value Decomposition (SVD)
  - t-distributed Stochastic Neighbor Embedding (TSNE)

Principle Component Analysis (PCA)

- Principal component analysis (PCA) is the process of computing the principal components and using them to perform a change of basis on the data.
- PCA is used in exploratory data analysis and for making predictive models.
- We use only the first few principal components and ignore the rest, so it is used for dimension reduction.
- It is used for dimensionality reduction by projecting each data point onto only the first few principal components to obtain lower-dimensional data while preserving as much of the data's variation as possible.

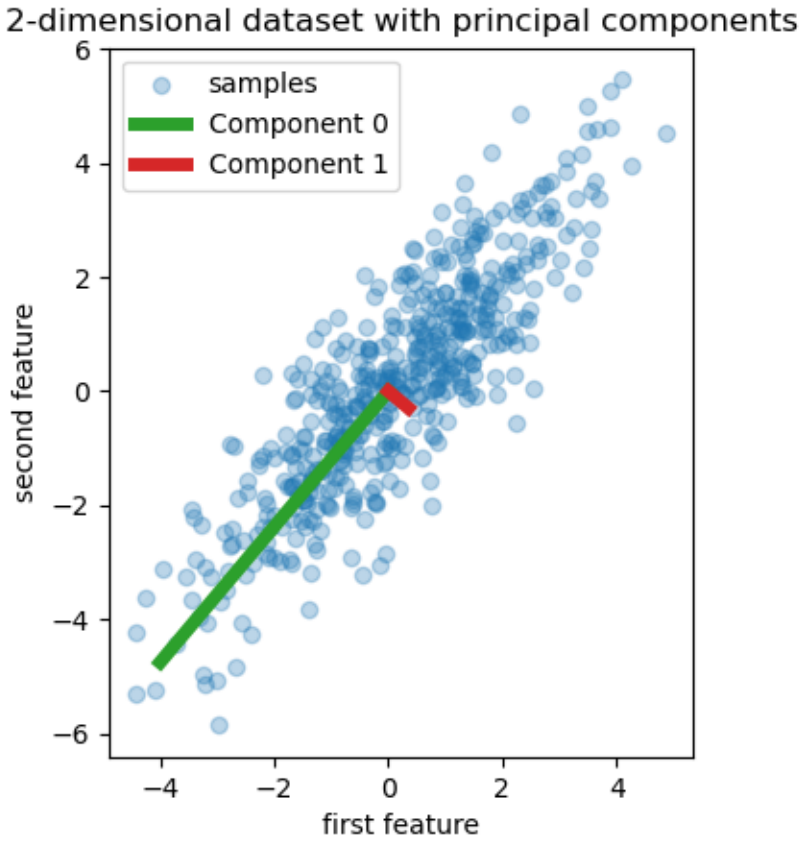
Principle Component Analysis (PCA)



Source: Lavrenko and Sutton 2011

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
rng = np.random.RandomState(0)
n_samples = 500
cov = [[3, 3], [3, 4]]
X = rng.multivariate_normal(mean=[0, 0], cov=cov, size=n_samples)
pca = PCA(n_components=2).fit(X)
plt.scatter(X[:, 0], X[:, 1], alpha=0.3, label="samples")
```

```
for i, (comp, var) in enumerate(
    zip(pca.components_, pca.explained_variance_)):
    comp = comp*var #scale component by its variance explanation power
    plt.plot(
        [0, comp[0]],
        [0, comp[1]],
        label=f"Component {i}",
        linewidth=5,
        color=f"C{i + 2}",
    )
plt.gca().set(
    aspect="equal",
    title="2-dimensional dataset with principal components",
    xlabel="first feature",
    ylabel="second feature",
)
plt.legend()
plt.show()
```

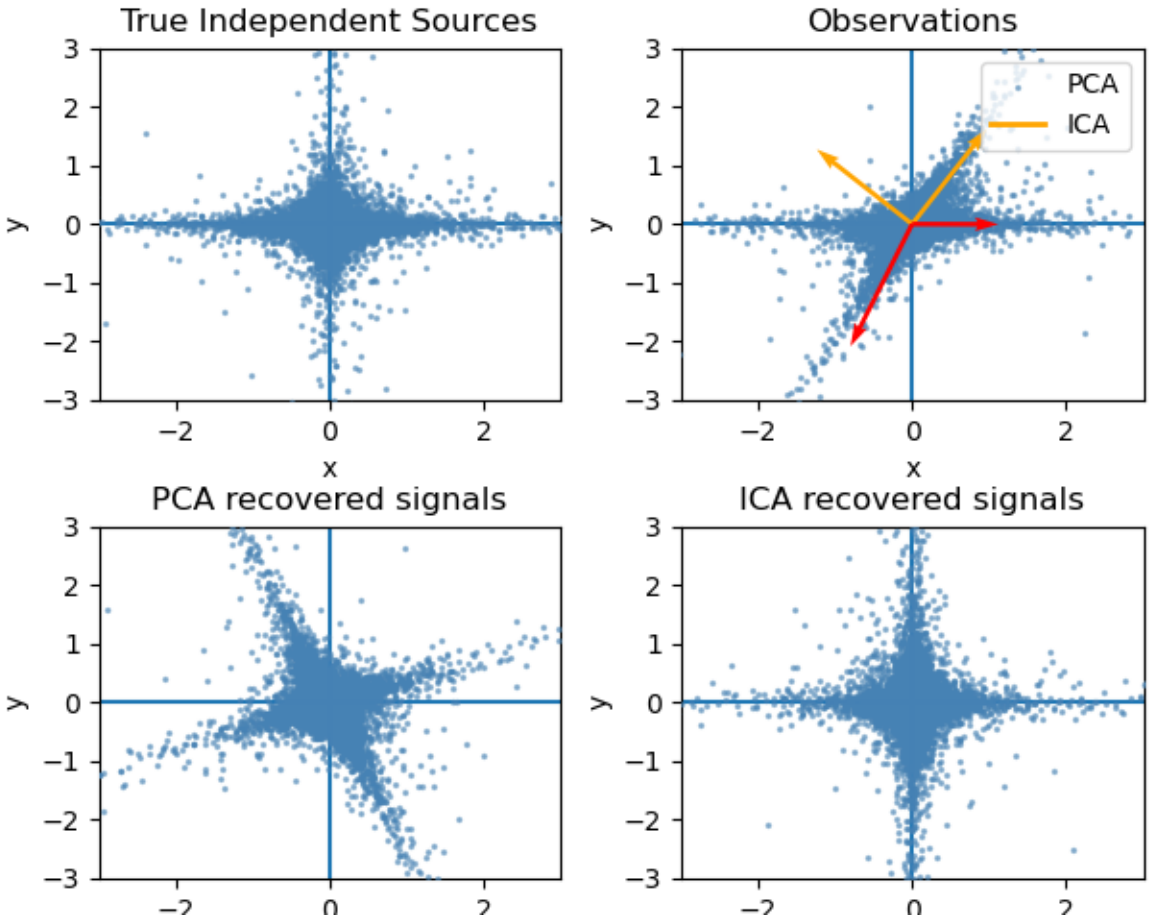

2-dimensional dataset with principal components
samples
Component 0
Component 1
second feature
first feature
6
4
2
0
−2
−4
−6
−4
−2
0
2
4

Independent component analysis (ICA)

- Independent component analysis (ICA) is a statistical and computational technique for revealing hidden factors that underlie sets of random variables, measurements, or signals.
- ICA defines a generative model for the observed multivariate data, which is typically given as a large database of samples.
- In the model, the data variables are assumed to be linear mixtures of some unknown latent variables, and the mixing system is also unknown.
- The latent variables are assumed non-gaussian and mutually independent, and they are called the independent components of the observed data.
- These independent components, also called sources or factors, can be found by ICA.

Independent Component Analysis (ICA)

The data are represented by the observed random vector $\boldsymbol{x} = (x_1, \ldots, x_m)^T$ and the hidden components as the random vector $\boldsymbol{s} = (s_1, \ldots, s_n)^T$. The task is to transform the observed data $\boldsymbol{x}$, using a linear static transformation $\boldsymbol{W}$ as $\boldsymbol{s} = \boldsymbol{W}\boldsymbol{x}$, into a vector of maximally independent components $\boldsymbol{s}$ measured by some function $F(s_1, \ldots, s_n)$ of independence.


True Independent Sources
Observations
PCA
ICA
PCA recovered signals
ICA recovered signals
x
y

PCA vs ICA

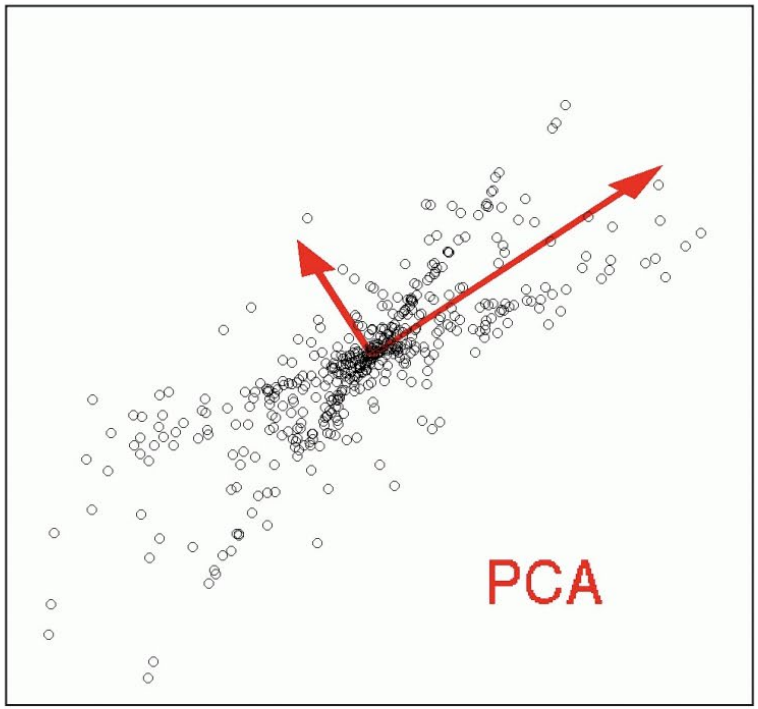
PCA:

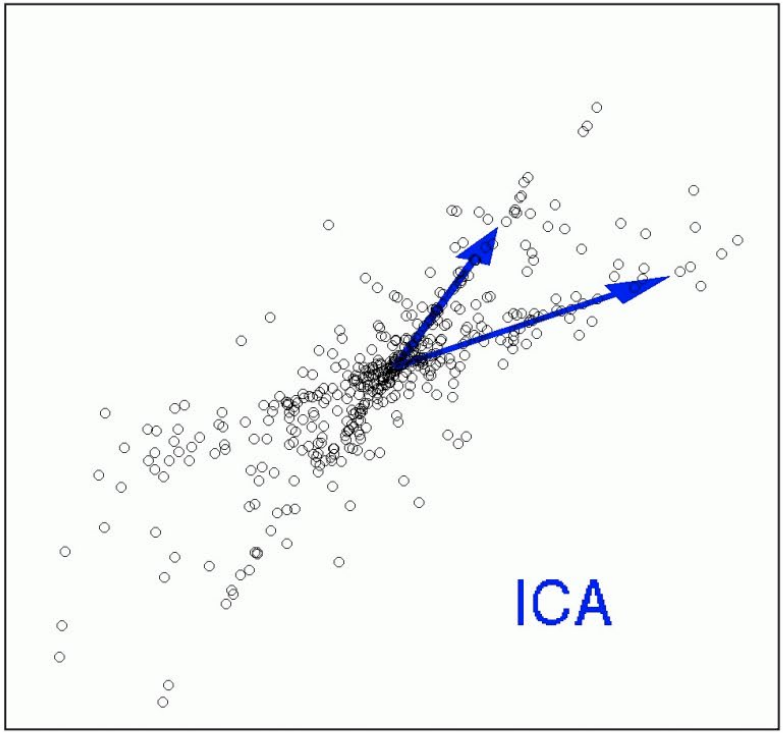
1. Find projections to minimize reconstruction error
   - Variance of projected data is as large as possible
2. 2nd-order statistics needed (cov(x))

ICA:

1. Find "interesting" projections
   - Projected data look as non-Gaussian, independent as possible
2. Higher-order statistics needed to measure degree of independence

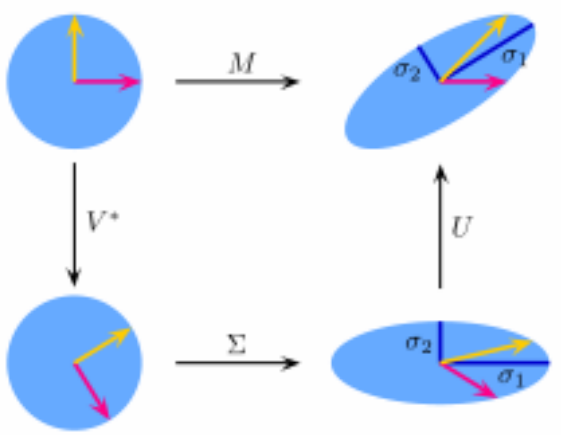
PCA vs ICA





Singular Value Decomposition (SVD)

- The singular value decomposition (SVD) is a factorization of a real or complex matrix.
- It generalizes the eigendecomposition of a square normal matrix with an orthonormal eigenbasis to any $n \times n$ matrix.
- It is related to the polar decomposition.



$$M = U \cdot \Sigma \cdot V^*$$

SVD Calculation

An **eigenvector** of a square matrix $\mathbf{A}$ is a vector $\mathbf{v}$ such that $\mathbf{A}$ only changes the magnitude of $\mathbf{v}$

- I.e. $\mathbf{Av} = \lambda\mathbf{v}$ for some $\lambda \in \mathbb{R}$
- Such λ is an **eigenvalue** of $\mathbf{A}$

The **eigendecomposition** of $\mathbf{A}$ is $\mathbf{A} = \mathbf{Q}\boldsymbol{\Delta}\mathbf{Q}^{-1}$

- The columns of $\mathbf{Q}$ are the eigenvectors of $\mathbf{A}$
- Matrix $\boldsymbol{\Delta}$ is a diagonal matrix with the eigenvalues

Not every (square) matrix has eigendecomposition

- If $\mathbf{A}$ is of form $\mathbf{BB}^T$, it always has eigendecomposition

The SVD of $\mathbf{A}$ is closely related to the eigendecompositions of $\mathbf{AA}^T$ and $\mathbf{A}^T\mathbf{A}$

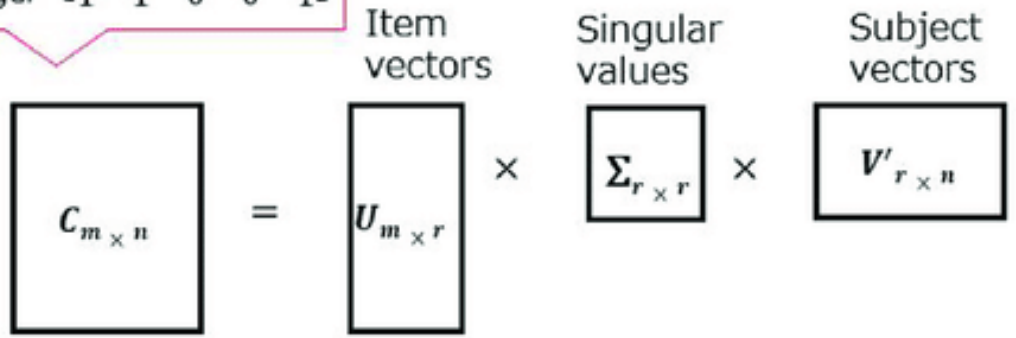
- The left singular vectors are the eigenvectors of $\mathbf{AA}^T$
- The right singular vectors are the eigenvectors of $\mathbf{A}^T\mathbf{A}$
- The singular values are the square roots of the eigenvalues of both $\mathbf{AA}^T$ and $\mathbf{A}^T\mathbf{A}$

SVD for Dimension Reduction

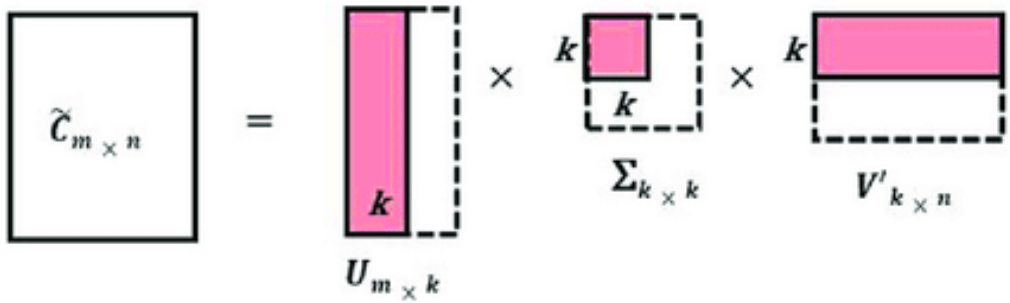
Item x subject matrix (ISM)

	S1	S2	S3	S4	S5
dog	1	1	1	1	1
cat	1	1	0	1	0
cow	0	0	1	0	1
lion	0	0	1	1	0
tiger	1	1	0	0	1

Singular decomposition analysis (SVD)



Reducing dimensions from r to k



t-distributed Stochastic Neighbor Embedding (TSNE)

- A method for visualizing high-dimensional data by giving each datapoint a location in a two or three-dimensional map.
- It is based on Stochastic Neighbor Embedding originally developed by Sam Roweis and Geoffrey Hinton, where Laurens van der Maaten proposed the t-distributed variant.
- It is a nonlinear dimensionality reduction technique well-suited for embedding high-dimensional data for visualization in a low-dimensional space of two or three dimensions.
- It models each high-dimensional object by a two- or three-dimensional point in such a way that similar objects are modeled by nearby points and dissimilar objects are modeled by distant points with high probability.

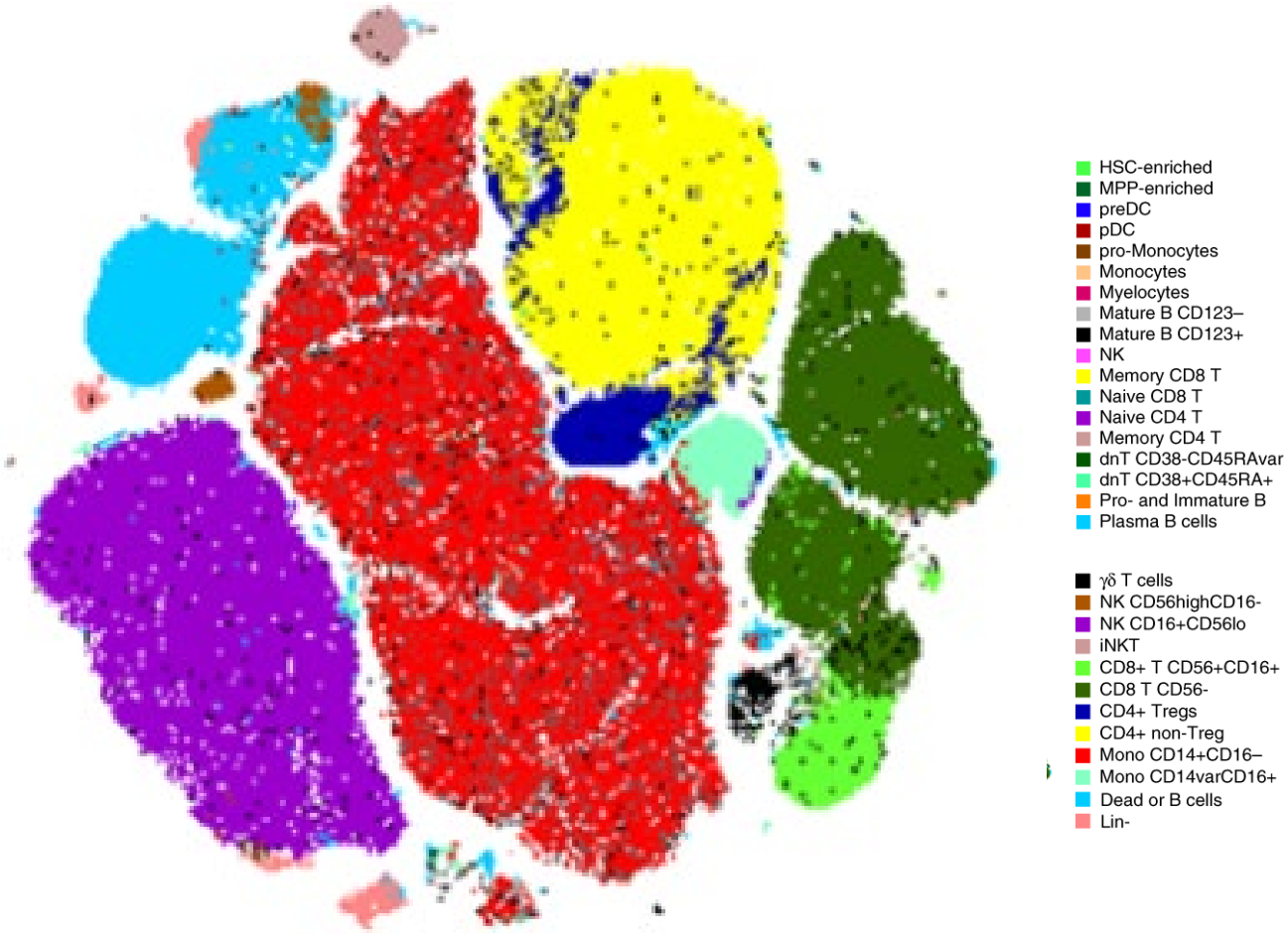
t-SNE

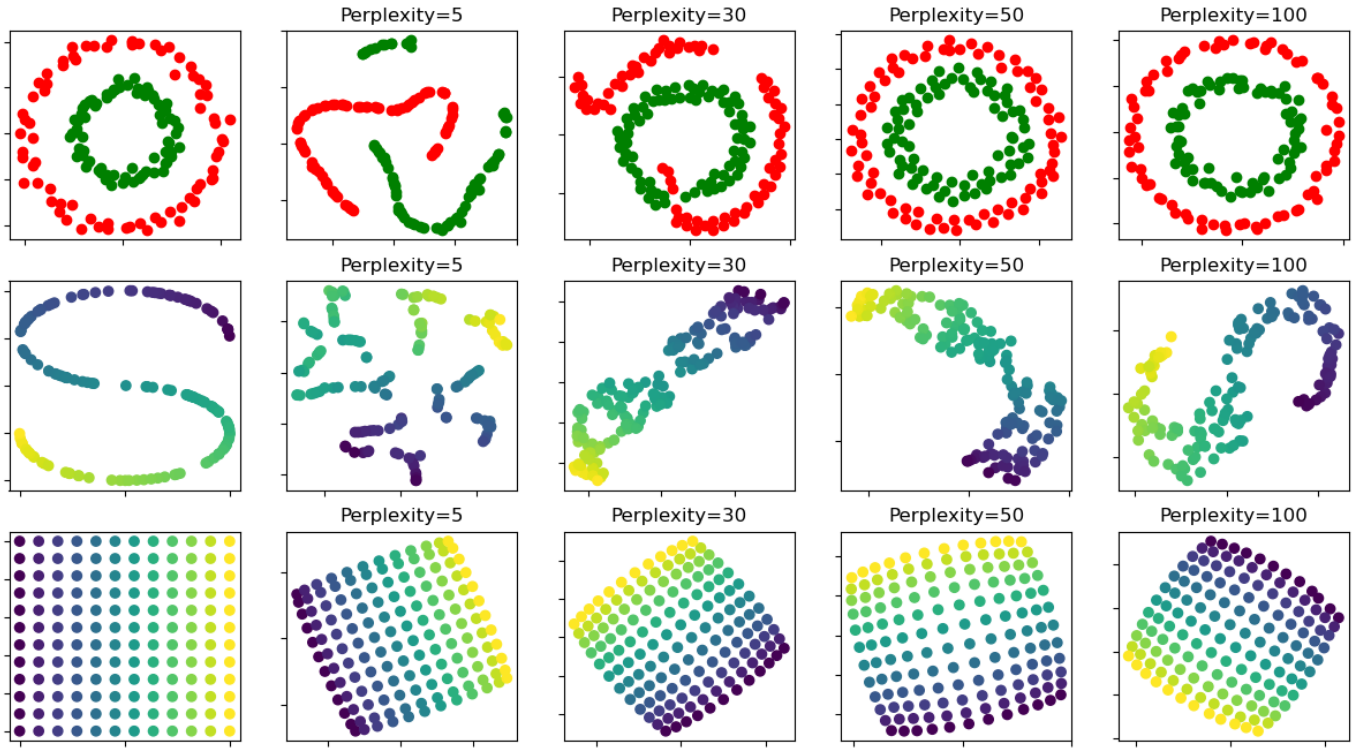
- The t-SNE algorithm comprises two main stages.

- **1st:**
  - t-SNE constructs a probability distribution over pairs of high-dimensional objects in such a way that similar objects are assigned a higher probability while dissimilar points are assigned a lower probability.

- **2nd:**
  - t-SNE defines a similar probability distribution over the points in the low-dimensional map, and it minimizes the Kullback–Leibler divergence (KL divergence) between the two distributions with respect to the locations of the points in the map.

t-SNE


HSC-enriched
MPP-enriched
preDC
pDC
pro-Monocytes
Monocytes
Myelocytes
Mature B CD123–
Mature B CD123+
NK
Memory CD8 T
Naive CD8 T
Naive CD4 T
Memory CD4 T
dnT CD38-CD45RAvar
dnT CD38+CD45RA+
Pro- and Immature B
Plasma B cells
γδ T cells
NK CD56highCD16-
NK CD16+CD56lo
iNKT
CD8+ T CD56+CD16+
CD8 T CD56-
CD4+ Tregs
CD4+ non-Treg
Mono CD14+CD16–
Mono CD14varCD16+
Dead or B cells
Lin-


Perplexity=5
Perplexity=30
Perplexity=50
Perplexity=100
Perplexity=5
Perplexity=30
Perplexity=50
Perplexity=100
Perplexity=5
Perplexity=30
Perplexity=50
Perplexity=100

Optimizing Parameters

GridSearchCV

- sklearn.model_selection.GridSearchCV(estimator, param_grid, scoring=None, n_jobs=None, iid='warn', refit=True, cv='warn', verbose=0, pre_dispatch='2*n_jobs', error_score='raise-deprecating', return_train_score=False)

GridSearchCV

decision_function(self, X)	Call decision_function on the estimator with the best found parameters.
fit(self, X[, y, groups])	Run fit with all sets of parameters.
get_params(self[, deep])	Get parameters for this estimator.
inverse_transform(self, Xt)	Call inverse_transform on the estimator with the best found params.
predict(self, X)	Call predict on the estimator with the best found parameters.
predict_log_proba(self, X)	Call predict_log_proba on the estimator with the best found parameters.
predict_proba(self, X)	Call predict_proba on the estimator with the best found parameters.
score(self, X[, y])	Returns the score on the given data, if the estimator has been refit.
set_params(self, **params)	Set the parameters of this estimator.
transform(self, X)	Call transform on the estimator with the best found parameters.

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report
from sklearn.svm import SVC
# Loading the Digits dataset
digits = datasets.load_digits()
# To apply an classifier on this data, we need to flatten the image, to
# turn the data in a (samples, feature) matrix:
n_samples = len(digits.images)
X = digits.images.reshape((n_samples, -1))
y = digits.target
# Split the dataset in two equal parts
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5,
random_state=0)
```

```
# Set the parameters by cross-validation
tuned_parameters = [
    {"kernel": ["rbf"], "gamma": [1e-3, 1e-4], "C": [1, 10, 100, 1000]},
    {"kernel": ["linear"], "C": [1, 10, 100, 1000]},
]

scores = ["precision", "recall"]
```

```
for score in scores:
    print("# Tuning hyper-parameters for %s" % score)
    print()
    clf = GridSearchCV(SVC(), tuned_parameters, scoring="%s_macro" % score)
    clf.fit(X_train, y_train)
    print("Best parameters set found on development set:")
    print()
    print(clf.best_params_)
    print()
    print("Grid scores on development set:")
    print()
    means = clf.cv_results_["mean_test_score"]
    stds = clf.cv_results_["std_test_score"]
    for mean, std, params in zip(means, stds, clf.cv_results_["params"]):
        print("%0.3f (+/-%0.03f) for %r" % (mean, std * 2, params))
    print()
    print("Detailed classification report:")
    print()
    print("The model is trained on the full development set.")
    print("The scores are computed on the full evaluation set.")
    print()
    y_true, y_pred = y_test, clf.predict(X_test)
    print(classification_report(y_true, y_pred))
    print()
```

Pipelining

- Pipeline of transforms with a final estimator.
- Sequentially apply a list of transforms and a final estimator.
- Intermediate steps of the pipeline must be ‘transforms’, that is, they must implement fit and transform methods.
- The final estimator only needs to implement fit. The transformers in the pipeline can be cached using memory argument.
- The purpose of the pipeline is to assemble several steps that can be cross-validated together while setting different parameters.

Pipelining

Parameters:	**steps** : *list,* List of (name, transform) tuples (implementing fit/transform) that are chained, in the order in which they are chained, with the last object an estimator. **memory** : *None, str or object with the joblib.Memory interface, optional*Used to cache the fitted transformers of the pipeline. By default, no caching is performed. If a string is given, it is the path to the caching directory. Enabling caching triggers a clone of the transformers before fitting. Therefore, the transformer instance given to the pipeline cannot be inspected directly. Use the attribute named_steps or steps to inspect estimators within the pipeline. Caching the transformers is advantageous when fitting is time consuming. **verbose** : *boolean, optional*If True, the time elapsed while fitting each step will be printed as it is completed.
Attributes:	**named_steps** : *bunch object, a dictionary with attribute access*Read-only attribute to access any step parameter by user given name. Keys are step names and values are steps parameters.

Pipelining: Methods

- decision_function(self, X) Apply transforms, and decision_function of the final estimator
- fit(self, X[, y]) Fit the model
- fit_predict(self, X[, y]) Applies fit_predict of last step in pipeline after transforms.
- fit_transform(self, X[, y]) Fit the model and transform with the final estimator
- get_params(self[, deep]) Get parameters for this estimator.
- predict(self, X, **predict_params) Apply transforms to the data, and predict with the final estimator
- predict_log_proba(self, X) Apply transforms, and predict_log_proba of the final estimator
- predict_proba(self, X) Apply transforms, and predict_proba of the final estimator
- score(self, X[, y, sample_weight]) Apply transforms, and score with the final estimator
- set_params(self, **kwargs) Set the parameters of this estimator.

Pipeline Example

```
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline

X, y = make_classification(random_state=101)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=101)

pipe = Pipeline([('scaler', StandardScaler()), ('svc', SVC())])
# The pipeline can be used as any other estimator
# and avoids leaking the test set into the train set
pipe.fit(X_train, y_train)
pipe.score(X_test, y_test)
```