Chapter 1 Introduction

A. Introduction

1. Sample / Instance: Every data point.
2. Feature: To describe these data points.

B. Classification:

1. Supervised learning: learn from cases we already know the answers，並從中得知結果，輸出結果為已知選項之一
2. Unsupervised learning: 只有輸入數據已知，其餘皆為未知

C. **Model Evaluation:** 利用 training data & test data

Chapter 2 Supervised Learning

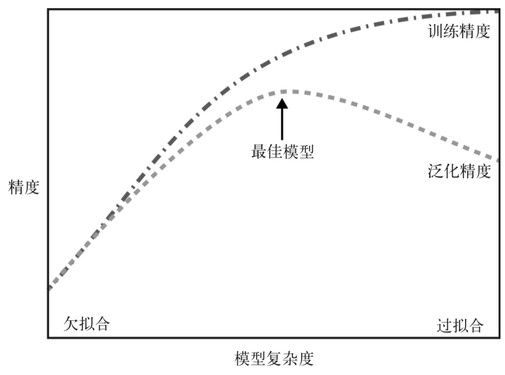
Model	Class/ Reg.	Code	Knob	Pre handle
K neighbor	C	from sklearn.neighbors import KNeighborsClassifier	n_neighbors	Y
Ordinary Least Square (OLS)	R	sklearn.linear_model import LinearRegressior	None	N
Ridge	R	from sklearn.linear_model import Ridge	alpha ⬆️generalize ⬆️	N
Lasso	R	from sklearn.linear_model import Lasso	alpha ⬆️regulation ⬆️	N
Decision Tree	C/R	from sklearn.tree import DecisionTreeRegressor from sklearn.tree import DecisionTreeClassifier	max_depth, max_leaf_nodes, min_samples_leaf	N
Random Forest	C/R	from sklearn.ensemble import RandomForestClassifier	n_estimators max_features max_depth, max_leaf_nodes	N
Gradient Boosted Decision Tree	C/R	from sklearn.ensemble import GradientBoostingClassifier	learning_rate n_estimators Max_depth max_leaf_nodes	N
SVM		from sklearn.svm import SVC	gamma and C ⬆️, complicated model.	Y
Deep Learning		from sklearn.neural_network import **MLPClassifier**	Hidden layer amount and size.⬇️, model complication ⬇️. Nonlinear function relu or tahn. L2 alpha.⬆️model complication ⬇️.	Y

A. Supervised learning includes “classification” & “regression”

1. Classification: 目標預測分類結果
2. Regression: 預測一個連續值，以coding language 來說是float number, aka.real number in math. Eg. 以教育程度預測年收入

B. 數據預測

1. Generalize: 一個模型可以預測他沒看過的數據，稱為 generalize
2. Overfitting: 用已知數據建造一個過於複雜的模型，導致模型在 training data 表現很好，但 generalize 不好。這種現象通常模型在 training data 的精準度 >> test data
3. Underfitting: 用已知數據建造的模型過於簡單，導致模型 generalize 能力差。這種現象通常模型在預測 training data & test data 的精準度都差



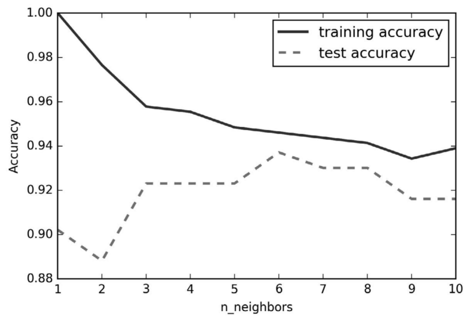
C. 預測模型

I. **<u>K neighbor</u>**

1. Definition: 只考慮最近鄰，也就是離預測數據點最近的 training 數據為預測結果。可以改變 neighbor 數量，結果以多數決決定。
2. Code:

```
from sklearn.neighbors import KNeighborsClassifier
Data = KNeighborsClassifier(n_neighbors= 3) #用離預測點最近的三個數據點做預測
Data.score(X_test, y_test) #評估模型預測好壞
```

3. 模型調整：
   1. 參數： neighbor 數量
   2. 意義： 隨著 neighbor 數量越大，decision boundary 越平滑，模型趨向簡單。極端情況，neighbor 個數 = training data數量，則每次預測的結果會一樣（就是training data 中，出現最多次的類別）



4. Knn for Regression Problems

Code:

```
from sklearn.neighbors import KNeighborsRegresson
reg = KNeighborsRegressor(n_neighbors=3)
reg.fit(X_train, y_train)
reg.score(X_test, y_test) # Regression score is value
```

5. Pros and cons:
    1. Pros:
        1. Model is easy to interpret
        2. Fast
    2. Cons:
        1. 數據預處理很重要
        2. 對 Feature 多的數據預測效果不好 （eg. feature 數 > 100）
        3. 對大多數 Feature 的取值多為0的數據預測效果不好 aka. 稀疏數據 Sparse Data

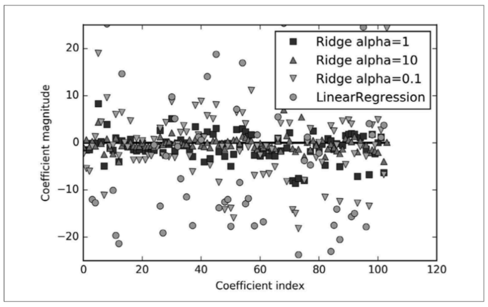
II. <u>Linear Model</u>

1. Definition: ŷ = w[0] * x[0] + w[1] * x[1] + ... + w[p] * x[p] + b
    a. x[0]~x[p] 代表 sample 的 feature （總共 p+1 個 feature）
    b. w & b 為模型參數
    c. y 為預測結果
    d. 如果只有一個 feature 的話：ŷ = w[0] * x[0] + b （w is slope, b is y-axis intercept)
    e. 以 regression 來說，一個 feature 時，預測結果為一條線，兩個 feature 時，預測結果為一個平面，超過兩個後，預測結果為超平面
2. 預測模型：
    1. Ordinary Least Square (OLS)
        a. Definition: 以參數 w & b 找出 (預測值-真實值) 的最小 mean square error (預測值-真實值）^2 / sample數）。OLS 沒有調整參數
        b. Code:

```
from sklearn.linear_model import LinearRegressior
Data = LinearRegression().fit(X_train, y_train)
# 查詢 w (slope) & b (intercept)
Data.coef_ #w 如果多個 feature,則為一個numbly 數列
Data.intercept_ #b 為一個 float number
Data.score(X_train, y_train)
Data.score(X_test, y_test))
```

2. **Ridge Regression**
   1. Definition: 預測公式與 ordinary least square 相同，但對 w 做更多約束。每個 w 值盡量趨近0，也就是說每個 feature 的影響力減小（斜率小），這種手法稱為Regulation（約束 w 以避免overfitting ）Ridge regression 用的是 L2 regulation
   2. Code:

```
from sklearn.linear_model import Ridge
    ridge = Ridge().fit(X_train, y_train)
    ridge.score(X_train, y_train)
    ridge.score(X_test, y_test)
```



   3. Model Adjusting knob:
      a. Knob: alpha. Defult = 1.0
      b. Meaning: alpha ⬆️, coefficient w closer to 0. Training function⬇️, generalize function ⬆️. (Ridge with smaller alpha is near to linear OLS)
      c. Remark: Ridge performance better training score than linear model in small data size. If the data size is big enough, the training score of ridge and linear model will be closer.

3. **Lasso Regression**
   1. Definition: Lasso is similar to ridge except for using L1 regulation instead of L2. L1 regulation resulting in some of the coefficients will be 0. This means that some of the features can be entirely ignored. This function brings out the most important features and makes the model easier to interpret.
   2. Code:

```
from sklearn.linear_model import Lasso
    lasso = Lasso().fit(X_train, y_train)
    lasso.score(X_train, y_train)
    lasso.score(X_test, y_test)
    np.sum(lasso.coef_ != 0) #Number of features used (Not 0 coefficient)
```

   3. Model Adjusting knob:
      a. alpha (Default = 1). By adjusting alpha, we need to increase “max_iter” at the same time.

b. Meaning: alpha ⬆️, regulation function ⬆️. If both testing and training score are low, the model is under fitting due to regulation is too powerful. In this case, number of features used will be small.

c. Code:

```
lasso001 = Lasso(alpha=0.01, max_iter=100000).fit(X_train, y_train)
```

d. Remark: In practical, we try to use ridge model in most cases but if we already know that there only few features are important, lasso model is a better chose.

III. <u>Linear Model for Classification Question</u>

1. Definition: $\hat{y} = w[0] * x[0] + w[1] * x[1] + ...+ w[p] * x[p] + b > 0$
   a. Instead of having sum of the equation, we have a > or < 0 prediction.
   b. If <0, classified as -1, if >0, classified as +1.
   c. In classification problem, the y function is the decision boundary. The decision boundary can be a 2D line, 3D plane or above.
2. Model:
   1. **Logistic Regression:**
   2. **Linear Support Vector Machine (SVM)**
   3. Code:

      ```
      from sklearn.linear_model import LogisticRegression
      from sklearn.svm import LinearSVC
      for model, ax in zip([LinearSVC(), LogisticRegression()], axes):
              clf = model.fit(X, y)
      ```
   4. Model Adjusting Knob:
      a. Logistic L2 regulation “C” (Default = 1). C ⬆️, regulation function ⬇️ . C ⬇️model emphasize on more coefficient closer to 0.
      b. If model need easier to interpret, logistic model can be changed into L1 regulation.
      c. Code: lr_l1 = LogisticRegression(C=C, penalty="l1").fit(X_train, y_train)

IV. <u>Decision Tree</u>

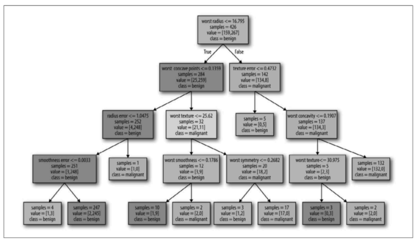
1. Definition: Decision tree is made of a series of if/else question. It can be applied on both regression and classification problem.
2. Model adjusting knob: Decision trees are easily overfitting. We can apply “pre-pruning”, “post-pruning (pruning)” to prevent over fitting. scikit-learn only has pre-pruning.
3. Code:

   ```
   from sklearn.tree import DecisionTreeRegressor
   ```

```
from sklearn.tree import DecisionTreeClassifier
    tree = DecisionTreeClassifier(random_state=0)
    tree.fit(X_train, y_train)
    tree.score(X_train, y_train)
    tree.score(X_test, y_test)
or tree = DecisionTreeClassifier(max_depth=4, random_state=0)
# If overfitting, set max_depth, max_leaf_nodes, min_samples_leaf
to limit if/else questions
```

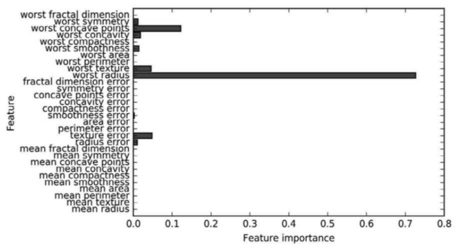
4. Model Analysis:
   a. Visualize decision tree

```
from sklearn.tree import export_graphviz
export_graphviz(tree, out_file=“tree.dot", class_names =
["malignant","benign"], feature_names = cancer.feature_names,
impurity = False, filled=True)

import graphviz
    with open("tree.dot") as f:
        dot_graph = f.read()
    graphviz.Source(dot_graph)
```



   b. Feature importance: Listed all the features according to their importance. The value is between 0 & 1. “0” means this feature is useless. The sum of all features is always 1.

```
tree.feature_importances_
def plot_feature_importances_cancer(model):
        n_features = cancer.data.shape[1]
        plt.barh(range(n_features), model.feature_importances_,
align='center')

plt.yticks(np.arange(n_features),
cancer.feature_names)
        plt.xlabel("Feature
importance")
        plt.ylabel("Feature")
    plot_feature_importances_cancer(tree)
```



5. Pros & cons:
   a. Pros: Model is easy to visualize and interpret. Not affected by feature sizes, no need to pre process data.
   b. Cons: Model is easy to be overfitting.

V. Decision Tree Ensemble

1. **Random Forest**
   1. Definition:
      a. Ensemble of many decision trees. Every tree has a little bit different. Every tree might predict better on its own but meanwhile over fitting. Thus random forest takes the average of all trees to prevent over fitting.
      b. Random forest can be applied on both regression and classification problem. The answer will be the average of all trees for regression problem and votes for classification.
   2. Building Random Forest:
      a. Bootstrap sample: We use n_samples to build a tree. (Same data can be picked repeatedly and randomly for n_samples times.) This will build a database not quite same as the original one but with same database size.
      b. Max Feature: We use max_feature to decide how many features in the original data are we using for the new decision tree. If max_feature = n_features, all the features will be considered. If max_feature ⬆️, the trees in random forest will be more alike. max_feature⬇️, there is more difference among all the trees and thus need more leaf(nodes) to fit the model.
      c. Code:

```
from sklearn.ensemble import RandomForestClassifier
    forest = RandomForestClassifier (n_estimators=5,
random_state=2)
# n_estimators=5 five trees.
# random_state=2, fixed if you want the same anwser
    forest.fit(X_train, y_train)
```

      d. Model Adjusting Knob:
         (1) n_estimators: The bigger the better.
         (2) max_features: Smaller max_features can prevent overfitting. But in experience, the best max_features is default. For classification, default = sqrt(n_features), for regression, default = n_features.
         (3) Pre-pruning: max_depth, max_leaf_nodes
2. **Gradient Boosted Decision Tree**
   1. Definition:
      a. Every tree can adjust the previous one. Using pre-pruning instead of random trees.
      b. Can be applied on both regression and classification problem.
   2. Model Adjusting Knob:

   a. Learning Rate: learning_rate means the intensity of every trees adjusting the previous one. Learning_rate ⬆️, model becomes more complicated. (Default = 0.1)
   b. N_estimators: n_estimators control the value of trees. n_estimators ⬆️, model is more complicated. (Default = 100, max_depth = 3)
   c. Max_depth (or max_leaf_nodes): In general, max_depth <= 5
3. Code:

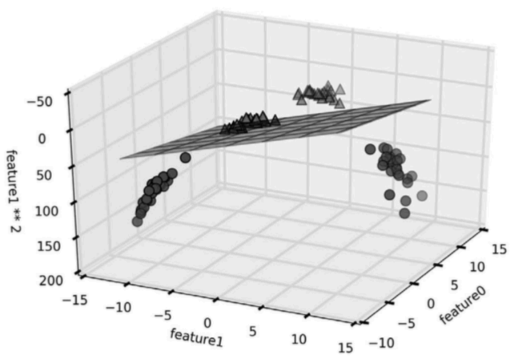
```
from sklearn.ensemble import GradientBoostingClassifier
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
 # If overfitting, method 1: adjust max_depth
    gbrt.fit(X_train, y_train)
    gbrt.score(X_train, y_train)
    gbrt.score(X_test, y_test)
# If overfitting, method 2: adjust learning rate
gbrt = GradientBoostingClassifier(random_state=0,
learning_rate=0.01)
    gbrt.fit(X_train, y_train)
```

4. Pros & Cons:
   a. Pros: One of the most used model. No need to normalize the database in previous.
   b. Cons: No good in high dimensional sparse data.

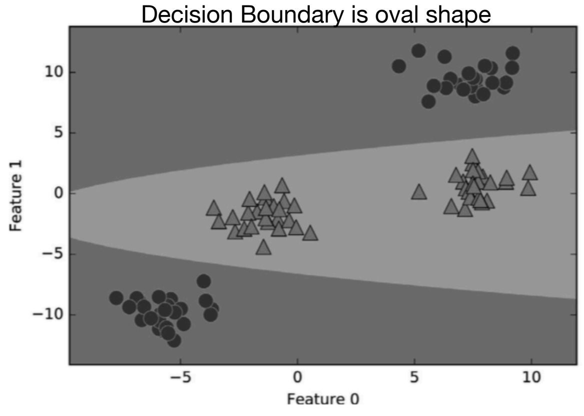
VI. <u>Kernelized Support Vector Machine (SVM)</u>

1. Definition:
   a. Add another feature by expanding original feature into higher dimension or polynomial. Eg. Change feature 1 into feature **2. Then, the original data point turn from 2D (feature 0, feature 1) into 3D (feature 0, feature 1, feature 1 **2)

Add (feature 1 **2), data point become 3D



Decision Boundary is oval shape

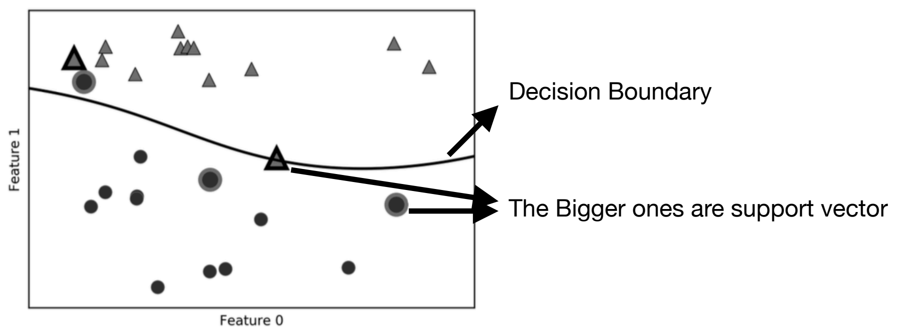


b. Support vector: Support vector means the data points near the decision boundary. New data prediction is based on the distance between new data and support vectors and the importance of support vectors. (Save in “dual_coef_”)

c. k_{rbf} (x1, x2) = exp $(-\gamma \| x1 - x2 \|^2)$ x1,x2 are data points. γ (gamma) decides the width of Gaussian core.

d. Code:

```
from sklearn.svm import SVC
X, y = mglearn.tools.make_handcrafted_dataset()
svm = SVC(kernel='rbf', C=10, gamma=0.1).fit(X, y)
mglearn.plots.plot_2d_separator(svm, X, eps=.5)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y) # Draw support vectors
sv = svm.support_vectors_
# The class of support vector is decide by dual_coef_+ or -
sv_labels = svm.dual_coef_.ravel() > 0
mglearn.discrete_scatter(sv[:, 0], sv[:, 1], sv_labels, s=15,
markeredgewidth=3) plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

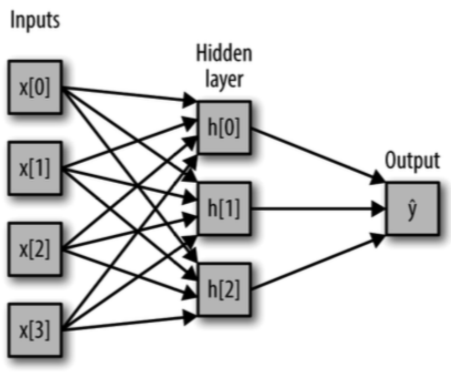


2. Model Tuning Knob:
   a. Gamma: Control Gaussian core. It decides the definition of “two data points are close.” Gamma ⬇️, Gaussian core ⬆️, more points are recognized as “close”. Gamma ⬇️, decision boundary change⬇️, model complicated ⬇️. Default = 1/n_features.
   b. C: Regulation coefficient. It limits the importance of each points. C ⬇️, the model is very limited, the influence of each points ⬇️. Default = 1.
   c. Both gamma and C ⬆️, means more complicated model. Use at same time.
3. Pros & Cons:

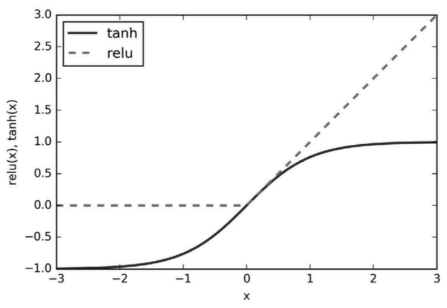
a. Pros: Performs well on both low/high dimensional database (few/many features).
b. Cons: Need data pre handle. Model needs to tune knob carefully.

VII. **Deep Learning**

1. Definition (Multilayer Perceptron MLP):
   a. MLP is recognized as linear model + series of hidden unit (so called unit layer) to calculate weighted sum and a non linear function, such as rectifying nonlinearity, relu and tangens hyperbolicus, tanh.
   b. relu cut off value under 0. tanh is closing to -1 when the input value is small, vise versa.



   c. $h[0] = \tanh(w[0, 0] * x[0] + w[1, 0] * x[1] + w[2, 0] * x[2] + w[3, 0] * x[3] + b[0])$
   $\hat{y} = v[0] * h[0] + v[1] * h[1] + v[2] * h[2] + b$
   w is the weighted between x and h.
   v is the weighted between h and output y.
   d. Number of hidden layers is a important parameter. For simple database, this parameter can be 10; for large database, this parameter can> 10000.



2. Coding:
   from sklearn.neural_network import **MLPClassifier**
   mlp = MLPClassifier(**solver='lbfgs', random_state=0**).fit(X_train, y_train)
3. Model Tuning Knob:
   a. Hidden layer amount and size.⬇️, model complication ⬇️.
   b. Nonlinear function relu or tahn. (Default = relu). Applied tahn has smoother decision boundary.
   c. L2 regulation alpha.⬆️, regulation function ⬆️. model complication ⬇️.
   d. Coding:
   mlp = MLPClassifier(solver='lbfgs', **activation='tanh',** random_state=0, **hidden_layer_sizes=[10, 10], alpha = 0.001**)
   # Two hidden layers, both size are 10.
4. Pros and Cons:
   a. Pros: Strong machine learning method
   b. Cons: long training period and need data pre handle.m

Chapter 3. Pre Process Data

A. Min Max Scaler

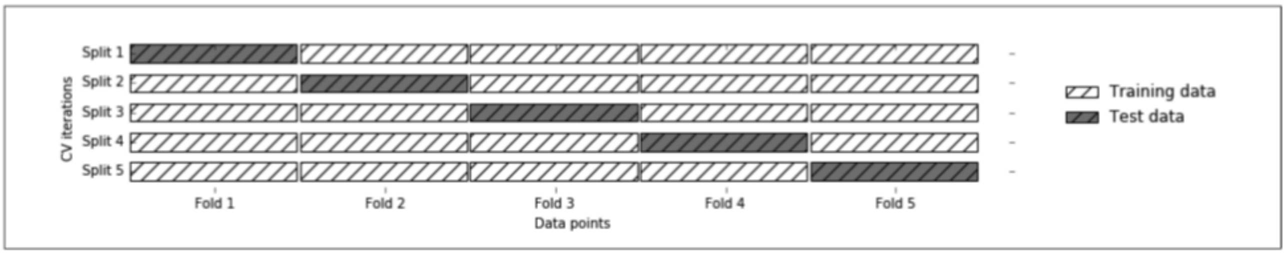
1. Definition: Move all data to make all features value within 0-1.
2. Code:

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
# "fit" calculate the Max and min value of all feature
scaler.fit(X_train)
# Use "transform" to rescale data
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
svm.fit(X_train_scaled, y_train)
```

Chapter 4. Model Evaluation and Improvement

A. Cross Validation

1. Definition: A statistical method to evaluate the generalized ability of a model. It separates databases into several portion and then every portion serves as training data in turn. These portion are called “fold”.



2. Cross validation in sckit learn

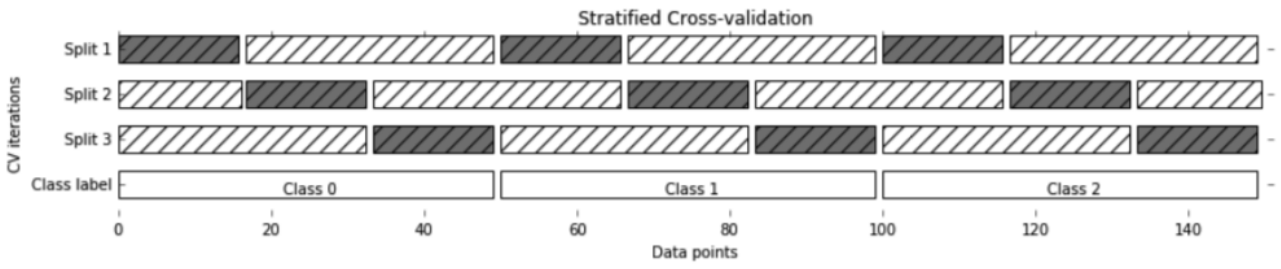
I. Cross validation

1) Code:

```
from sklearn.model_selection import cross_val_score
# Use “cv” to change fold number
scores = cross_val_score(logreg, iris.data, iris.target, cv=5)
```

II. Stratified k-fold cross-validation

1) Definition: The percentage of every class in a single fold are same as the whole database. For example, in the database, there are 80% A, 20% B. Every fold will keep this A, B ratio when split.



2) For classification problem, default is Stratified k-fold cross-validation, for regression problem, default is cross-validation.
3) Tuning knob:
   1. kfold (n_split)
      a. Definition: Use n_split to change cv amount.
      b. Code:

```
from sklearn.model_selection import KFold
kfold = KFold(n_splits=5)
cross_val_score(logreg, iris.data, iris.target, cv=kfold)
```

   2. kfold (shuffle)

a. Definition: Randomize database to avoid that there only a single class in one fold.
b. Code:

```
# set random_state value to assure same randomize data
kfold = KFold(n_splits=3, shuffle=True, random_state=0)
cross_val_score(logreg, iris.data, iris.target, cv=kfold)
```

III. Leave one out

1) Definition: Leave-one-out can be seen as there is only one data in every fold. There is only one data left for testing.
2) Code:

```
from sklearn.model_selection import LeaveOneOut
loo = LeaveOneOut()
scores = cross_val_score(logreg, iris.data, iris.target, cv=loo)
```

3) Pros & Cons: Need a lot of time to process but every effective for small size database.

B. Cross Validation Grid Search

1. Definition: Use grid search to find best model parameter (tuning knob) and apply cross validation.
2. Code:

```
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
              'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
X_train, X_test, y_train, y_test = train_test_split(
      iris.data, iris.target, random_state=0)
grid_search.fit(X_train, y_train)
grid_search.score(X_test, y_test)
```

3. Grid Search Analysis
   a. grid_search.best_params_: The best model parameter than grid search found.
   b. grid_search.best_score_: Best score in cross validation.
   c. grid_search.best_estimator_: The model that respond to the best parameter.
   d. grid_search.cv_results_: All the parameter and responded scores.

```
# Use data frame to show result for better reading.
results = pd.DataFrame(grid_search.cv_results_)
# Transform results for better reading
display(results.T)
```

4. Conditional grid search:
   a. Definition: SVC’s kernel can be set as linear or rbf. If kernel = linear, only parameter C matters. If kernel = rbf, C and gamma parameter matters. So when kernel = linear, no need to search parameter gamma.
   b. Code:

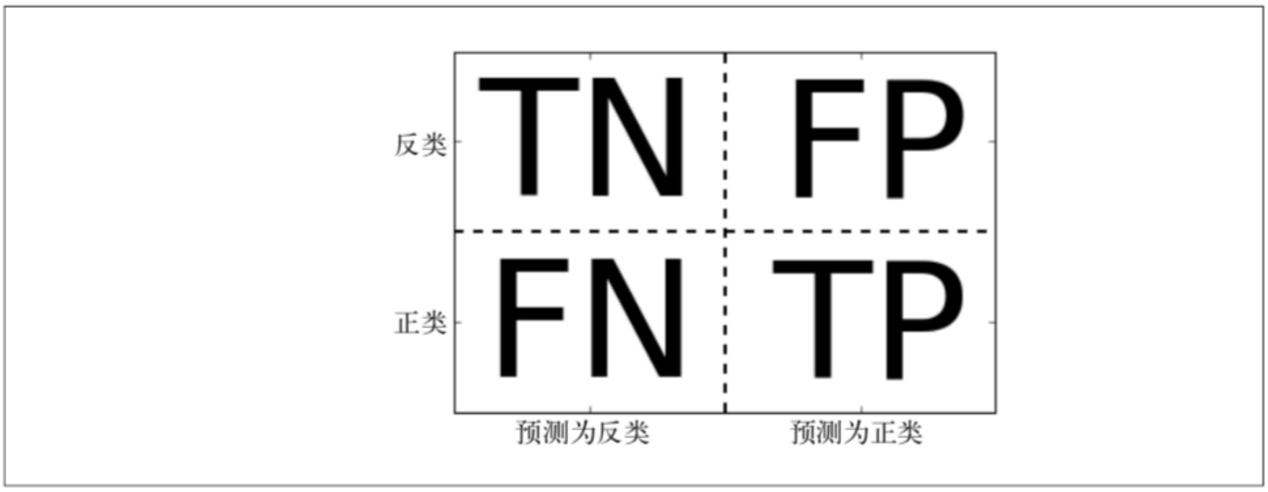
```
# A list of dictionaries
param_grid = [{'kernel': ['rbf'],
               'C': [0.001, 0.01, 0.1, 1, 10, 100],
               'gamma': [0.001, 0.01, 0.1, 1, 10, 100]},
              {'kernel': ['linear'],
               'C': [0.001, 0.01, 0.1, 1, 10, 100]}]
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
```

C. Two Class Validation Index

1. Definition:
   1) First we need to know our goal for machine learning prediction.
   2) Error types:
      a. False positive (Type 1 error): Predict a negative data as a positive class.
      b. False negative (Type 2 error): Predict a positive data as a negative class.
   3) Imbalanced dataset: A very common case in real world data. It means a dataset is full of positive (or negative) class data, and has very few negative (or positive) class data. And thus the test score we used to evaluate model precision is no longer presentive.
2. Confusion Matrix:
   1) For two-class classification, confusion matrix is the most effective way to evaluate the results.
   2) Code:

```
from sklearn.metrics import confusion_matrix
confusion = confusion_matrix(y_test, pred_logreg)
```

```
mglearn.plots.plot_binary_confusion_matrix()
```



3) Example:

```
# Dummy classifier predicts all data as the majority class of the dataset
from sklearn.dummy import DummyClassifier
dummy_majority = DummyClassifier(strategy='most_frequent').fit(X_train, y_train)
pred_most_frequent = dummy_majority.predict(X_test)
# Decision Tree
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier(max_depth=2).fit(X_train, y_train)
pred_tree = tree.predict(X_test)
# Logistic regression
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression(C=0.1).fit(X_train, y_train)
pred_logreg = logreg.predict(X_test)
# Dummy classifier predicts all data as random
dummy = DummyClassifier().fit(X_train, y_train)
pred_dummy = dummy.predict(X_test)
# Confusion matrix
confusion_matrix(y_test, pred_most_frequent)
confusion_matrix(y_test, pred_dummy)
confusion_matrix(y_test, pred_tree)
confusion_matrix(y_test, pred_logreg)
```

4. Confusion matrix analysis
   1) Accuracy: (TP + TN) / (TP + TN + FP + FN). aka. (TP+TB) / all samples.
   2) Precision (positive predictive value, PPV):
      a. Precision = TP / (TP + FP). How many dataset that is predicted as positive is truly positive.
      b. Example: Target is to limit the false positive. For example, a new drug is under evaluation, the company need to assure the drug is effective (positive) to do the human experiment.
   3) Recall (sensitivity, hit rate, true positive rate, TPR):
      a. Recall=TP/(TP + FN). How many data that is positive is predicted as positive.
      b. Example: Target is to avoid false negative. For example, the cancer disease is classify as positive. We need to find out all the patients that have cancer.
   4) f1 - score (f-measure):
      a. f1- score = 2* (precision * recall / (precision + recall)). Precision and recall is a trade off. f-score is the harmonic mean of precision and recall.
   5) Code: (Precision / recall / f1- score / support (true sample size under this class))

```
from sklearn.metrics import classification_report
print(classification_report(y_test, pred_most_frequent,
target_names=[“positive class name", “negative class name"]))
```

5. Decision function & predict proba
   1) Definition:
      a. Decision function: Mostly applied on SVM. decision_function default = 0. It means the data inside the “circle 0” will be classified as positive class.
      b. Predict proba: Mostly appled on decision tree. predict_proba default = 0.5. It means if the model has the confident larger than 0.5 to believe the data is positive, the data will be classified as positive class.
      c. Code:

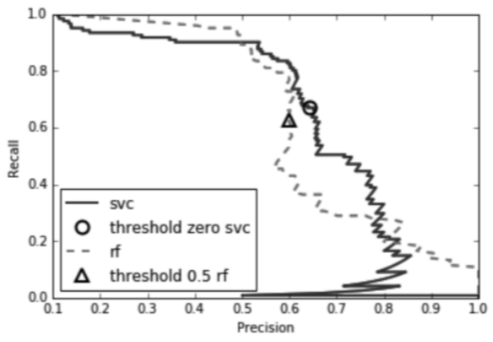
```
# Change decision function of SVM from 1 to 0.8.
y_pred_lower_threshold = svc.decision_function(X_test) > -.8
classification_report(y_test, y_pred_lower_threshold)
```

6. Precisoin-recall Curve
   1) Definition: Print out all the precision and recall value under different decision function or predict proba set point.
   2) Code:

```
# SVM with different decision function set point
from sklearn.metrics import precision_recall_curve
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
svc = SVC(gamma=.05).fit(X_train, y_train)
precision, recall, thresholds =
precision_recall_curve(y_test, svc.decision_function(X_test))
# Mark the point with decision function = 0
close_zero = np.argmin(np.abs(thresholds))
plt.plot(precision[close_zero], recall[close_zero], 'o', markersize=10,
          label="threshold zero", fillstyle="none", c='k', mew=2)
plt.plot(precision, recall, label="precision recall curve")
plt.xlabel("Precision")
plt.ylabel(“Recall")

# Random forest with different predict proba set point
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(n_estimators=100, random_state=0,
max_features=2)
rf.fit(X_train, y_train)
# The second parameter of precision_recall curve is the certainty of
positive class (class 1) so we use X_test[ : , 1]
precision_rf, recall_rf, thresholds_rf =
precision_recall_curve( y_test, rf.predict_proba(X_test)[:, 1])
```

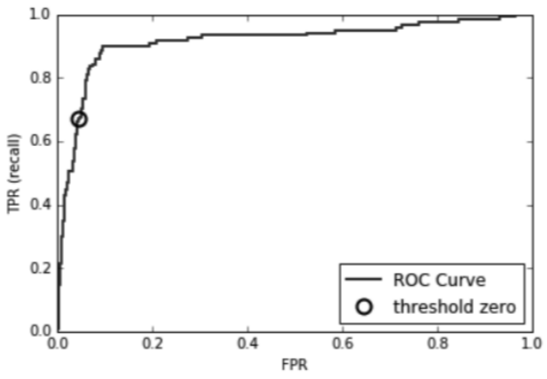
# Mark the point with predict proba = 0.6
close_default_rf = np.argmin(np.abs(thresholds_rf - 0.5))
plt.plot(precision_rf[close_default_rf], recall_rf[close_default_rf], '^', c='k',
markersize=10, label="threshold 0.5 rf", fillstyle="none", mew=2)

3) Average Precision Score:
   a. Definition: The area under precision_recall curve. Average precision ⬆️, better model.
   b. Code:



from sklearn.metrics import average_precision_score
# Average precision score for random forest
ap_rf = average_precision_score(y_test, rf.predict_proba(X_test)[:, 1])
# Average precision score for SVM
ap_svc = average_precision_score(y_test, svc.decision_function(X_test))

7. ROC and AUC
   1) Definition:
      a. Another way to evaluate the model performance under all decision function or predict proba set point. Similar to precision_recall curve.
      b. ROC = FP / (FP + TN). ROC present false positive rate (FPR) and true positive rate (TPR).
      c. Code:



from sklearn.metrics import roc_curve
fpr, tpr, thresholds = **roc_curve(y_test, svc.decision_function(X_test))**
plt.plot(fpr, tpr, label="ROC Curve")
plt.xlabel("FPR")
plt.ylabel("TPR (recall)")
# Mark the point with decision function = 0
close_zero = np.argmin(np.abs(thresholds))
plt.plot(fpr[close_zero], tpr[close_zero], 'o', markersize=10,
label="threshold zero", fillstyle="none", c='k', mew=2)
plt.legend(loc=4)

   2) AUC (Area under curve):
      a. Definition: The area under ROC curve. AUC ⬆️, better model.

b. Code:

```
from sklearn.metrics import roc_auc_score
# AUC score for random forest
rf_auc = roc_auc_score(y_test, rf.predict_proba(X_test)[:, 1])
# AUC score for SVM
svc_auc = roc_auc_score(y_test, svc.decision_function(X_test))
```

D. Multi Class Validation Index

1. Accuracy, confusion matrix and f1- score. Similar to two class validation index.
2. Code:

```
from sklearn.metrics import accuracy_score
X_train, X_test, y_train, y_test =
train_test_split(digits.data, digits.target, random_state=0)
lr = LogisticRegression().fit(X_train, y_train)
pred = lr.predict(X_test)
accuracy_score(y_test, pred)
confusion_matrix(y_test, pred)
# The columns present the true label, the rows present the predicted label
classification_report(y_test, pred)
```

3. F1-score
   1) F1- score is the most used validation method in multi class validation.
   2) F1 strategy:
      a. Marco: Non weighted mean for f1 score. It gives the same weight to all class, no matters the sample size of the class.
      b. Micro: Calculate the aggregations of FP, FN and TP in all class. And then calculate the precision, recall and f1 under these aggregations.
      c. Code:

```
f1_score(y_test, pred, average=“micro")
f1_score(y_test, pred, average="macro")
```

E. Regression Problem Validation Index

1. The default R^2 is the most used validation method.