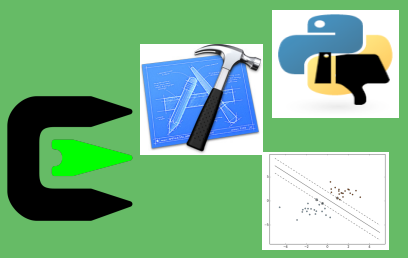
CISC 5352 FINANCIAL PROGRAMMING AND DATA ANALYTICS LECTURE NOTE (7)



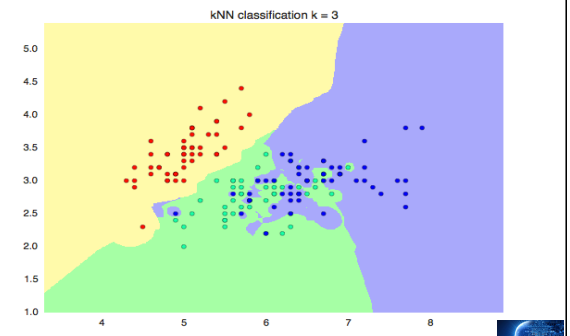
Henry Han Ph.D.
Department of Computer and Information Science
Fordham University, New York NY 10023

Last class review

① K-NN learning (sklearn module in **scikit-learn** package)
  ① k-NN classification
  ② K-NN regression
② Three steps: 1
  ① Get training/test data (there are different methods to get training and test data)
  ② Learning machine training : learn knowledge from known information→fit
    fit(training_data, training_data_label)
  ③ Predict: using the learning machine to predict test data (new data)→ predict
    test_data_label=predict(test_data)

```
from sklearn import neighbors

# 1. training and test data
training_data  = [[10], [11], [12], [30], [40], [88]]
training_label = [0, 0, 0, 1, 1, 1]

test_data =[[15.8], [98.38]]      # what to find labels of test data
#2. training

kNN = neighbors.KNeighborsClassifier(n_neighbors=3)
kNN.fit(training_data, training_label)

# 3. test/predict
test_data_label=kNN.predict(test_data)

print("The pedicted labels for test data is:\n")
print(test_data_label)
```

Last class review: data visualization in classification: How to do data visualization?



Last class review: data visualization in classification: How to do data visualization?

1. Create two related color maps (e.g. light, bold) for decision regions and training points
2. Sampling test data in a rectangle region generated by training data x_{min}, x_{max}, y_{min}, y_{max}
3. Do classification and label test data with different colors according to their predicted types
4. Plot training data in different color according to its types.

Note: Projects on two PCs are used for general data visualization

Last class review: support vector machines (SVM). We did not finish it completely

- Main idea: seek to construct an optimal hyperplane in a high-dimensional space with kernel trick learning tricks.
- It has a more intuitive geometric interpretation.

It classify data into three cases

1. linear separable case→ a hard margin hyperplane (no misclassification) $\min \frac{1}{2} w^t w \quad y_i(w^t x_i + b) \geq 1$
2. linear non-separable→ a soft margin hyperplane→ slack variables in optimization. (we allow some misclassification cases)
3. Nonlinear case → no hyperplane available at all because of nature of data nonlinearity: the relationship between data and its label is nonlinear
   1. Kernel tricks are used to handle nonlinear cases

It classify data into three cases

1. linear separable case→ a hard margin hyperplane (no misclassification). The hyperplane equation is $w^t x+b=0$

$\min \frac{1}{2} w^t w \quad y_i(w^t x_i + b) \geq 1$

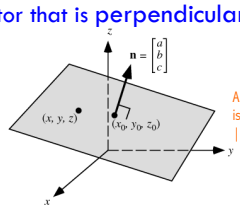
w is the norm vector of the hyperplane and b is an offset scalar

y_i is the label information for the i^{th} point (vector) x_i: it only has two values usually {-1,1}

Note: The plane in 3+ dimensional space is called a hyperplane

What's the norm of a plane?

A unit vector that is perpendicular to the plane



A unit vector is a vector whose norm is 1 $||n||=1$
$||n||=(a^2+b^2+c^2)^{1/2}=1$

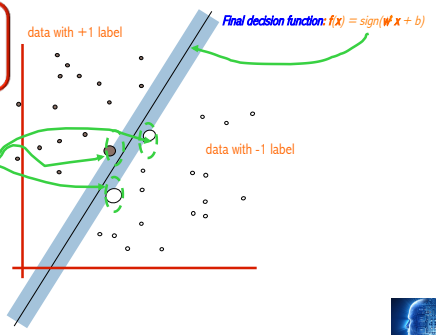
The equation of a plane can be determined by a normal vector n=(a,b,c) through the point $x_0=(x_0,y_0,z_0)$ is

$n\cdot(x-x_0)=0$ ('·' means inner product) It can also write as

$n^t(x-x_0)=0 \quad n^t x+(-n^t x_0)=0$ -→ **$w^t x+b=0$**

A hard margin: data are linear separable: no misclassification

$$\min \frac{1}{2} w'w$$
$$y_i(w'x_i + b) \geq 1$$



Support vectors are the decision makers for classification in SVM.

Using Lagrange multipliers α for each constraint in training data)

Find $\alpha_1 \ldots \alpha_N$ such that **Max** $\Sigma\alpha_i - \frac{1}{2}\Sigma\Sigma\alpha_i\alpha_j y_i y_j \mathbf{x_i}^T\mathbf{x_j}$
Under the following conditions
(1) $\Sigma\alpha_i y_i = 0$
(2) $\alpha_i \geq 0$
N means we have n training points

Solution:

Each non-zero α_i indicates that corresponding $\mathbf{x_i}$ is a support vector

$\mathbf{w} = \Sigma\ \alpha_i y_i \mathbf{x_i}$ $\quad b = y_k - \mathbf{w}^T\mathbf{x_k}$ for any $\mathbf{x_k}$ such that $\alpha_k \neq 0$

Decision function for a new point x

$f(\mathbf{x}) = \mathbf{W}^T\mathbf{x} + b = \Sigma\ \alpha_i y_i \mathbf{x_i}^T\mathbf{x} + b$

2. linear non-separable: soft Margin

Slack variables ξ_i can be added to allow misclassification of difficult or noisy examples. The corresponding optimization problem (L^1 –norm SVM) is specified as

Find $\mathbf{w}$ and b such that
$\min \Phi(\mathbf{w}) = \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\Sigma\ \xi_i$

such that for all i
$y_i(\mathbf{w}^T\mathbf{x_i} + b) \geq 1 - \xi_i$
$\xi_i \geq 0$

The penalty parameter C is the weight for slack variables
The larger the C value, the a stricter separation between classes that the optimization attempts to make.

How to select the penalty parameter C?

How to select the penalty parameter C?

The default is 1

The input data will be normalized to zero mean and 1 standard deviation data

How about nonlinear data?

We can't just see we can separate or not→their nonlinear property prevents the "separable" in input space.

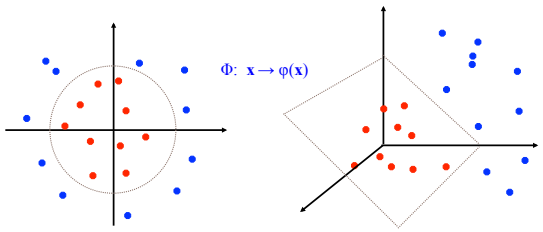
The best contribution of SVM to machine learning is to solve this problem using kernel tricks

SVM maps nonlinear data to a high-dimensional feature space to conduct classification

We don't need to know the mapping function Φ but we can still do classification in the high-dimensional space.

SVM maps nonlinear data to a high-dimensional feature space to conduct classification

We don't need to know the mapping function Φ but we can still do classification in the high-dimensional space.



The high-dimensional space does not bring high complexity. All computing can be done in input space via using kernel tricks.

Kernel tricks

Any input data point is mapped into high-dimensional space via a transformation $\Phi: x \rightarrow \phi(x)$.

We can use a kernel function to evaluate computing in the feature space by assume all computing can be written in inner-product forms.

$$K(x_i,x_j) = \phi(x_i)^T \phi(x_j)$$

The correctness of such an approach can be guaranteed by Mercer's theorem in functional analysis.

A kernel matrix must be a semi-positive definite matrix.

Positive definite matrix: a symmetric matrix whose eigenvalues are positive

Semi-positive definite matrix: symmetric matrix whose eigenvalues are >=0

General Kernel Functions

Linear: $K(\mathbf{x}_i,\mathbf{x}_j)= \mathbf{x}_i^T\mathbf{x}_j$

Polynomial of power p: $K(\mathbf{x}_i,\mathbf{x}_j)= (1+ \mathbf{x}_i^T\mathbf{x}_j)^p$ ('poly' kernel)

Gaussian ('rbf' kernel):

$$K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma^2})$$

Sigmoid: $K(\mathbf{x}_i,\mathbf{x}_j)= \tanh(\beta_0\mathbf{x}_i^T\mathbf{x}_j + \beta_1)$ ('sigmoid' kernel)

Note: you can build you own kernels if the corresponding kernel matrices are positive definite or Semi positive definite.

Which kernel should I use?

Which kernel should I use?

It may rely on your data and your 'experience'.

Linear kernel assumes that data is linear separable or linear non-separable.

Nonlinear kernels: 'rbf', 'poly', 'sigmoid'

For nonlinear data (e.g. image/financial data), nonlinear kernels are recommended.

Nonlinear SVM

Dual problem formulation:

Find $\alpha_1 \ldots \alpha_N$ such that
$Q(\alpha) = \Sigma\alpha_i - \frac{1}{2}\Sigma\Sigma\alpha_i\alpha_j y_i y_j K(x_i, x_j)$ is maximized and
(1) $\Sigma\alpha_i y_i = 0$
(2) $\alpha_i \geq 0$ for all α_i

Decision function of nonlinear SVM for a new data point x:

$$f(x) = \Sigma\ \alpha_i y_i K(x_i, x) + b$$

```
from sklearn import svm
import time

## training
X = [[0, 0], [-2,0], [1, 1], [10,1]]
## training data label
y = [0,0,1,1]
## training by using default parameter setting
clf = svm.SVC()
clf.fit(X, y)

## prediction for test data
predicted_label=clf.predict([[2.5, 8.]])

print("\n The predicted label is--->" +str(predicted_label))
time.sleep(1)
print("\n Checking parameter setting")
print(clf)
time.sleep(2)
```

```
from sklearn import svm
import time

## training
X = [[0, 0], [-2,0], [1, 1], [10,1]]
## training data label
y = [0,0,1,1]
## training by using default parameter setting
clf = svm.SVC()
clf.fit(X, y)

## prediction for test data
predicted_label=clf.predict([[2.5, 8.]])

print("\n The predicted label is--->" +str(predicted_label))
time.sleep(1)
print("\n Checking parameter setting:")
print(clf)
time.sleep(2)
```

```
The predicted label is--->[1]

 Checking parameter setting:

SVC(C=1.0,
cache_size=200,
class_weight=None,
coef0=0.0,
decision_function_shape=None,
degree=3,
gamma='auto',
kernel='rbf',
 max_iter=-1,
probability=False,
random_state=None,
shrinking=True,
 tol=0.001,
verbose=False  )
```

Kernels in SVC: 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'

'linear': (x, x') → inner product between x and x' → $x^t x'$

'poly' $(\gamma\langle x, x'\rangle + r)^d$.

- d is specified by parameter: 'degree' (default is 3)
- r is specified by parameter: 'coef0' (default is 0)

'rbf': $\exp(-\gamma|x - x'|^2)$.

γ is specified by parameter gamma (>0) (e.g. ½)

'sigmoid': $(\tanh(\gamma\langle x, x'\rangle + r))$, where r is specified by `coef0`.

Note: If gamma is 'auto' then it is set as 1/N, where N is number of training points

Modify the previous codes such that

1. Try all four kernels and find their corresponding predicted labels and corresponding support vectors (clf.support_vector_)
2. Set gamma=1/2
3. Set: Tol=0.0001

```
from sklearn import svm
import time

## training
X = [[0, 0], [-2,0], [1, 1], [10,1]]
## training data label
y = [0,0,1,1]
## test data
test_data = [[2.5, 8.]]

kernel_list=['linear', 'rbf', 'poly', 'sigmoid']

predicted_label = {}
support_vectors = {}

for kernel in kernel_list:
    clf = svm.SVC(kernel=kernel, tol=0.0001, gamma=0.5)
    clf.fit(X, y)
    predicted_label[kernel]=str(clf.predict(test_data))
    support_vectors[kernel]=clf.support_vectors_
```

```
print(" \nCheck the predicted labels under different kernels\n")
print(str(predicted_label))
time.sleep(2)

print(" \nCheck the support vectors under different kernels...\n")
print(str(support_vectors))
time.sleep(2)
```

```
print(" \nCheck the predicted labels under different kernels\n")
print(str(predicted_label))
time.sleep(2)

print(" \nCheck the support vectors under different kernels...\n")
print(str(support_vectors))
time.sleep(2)
```

Different kernels: same label But different support vectors!

```
{'sigmoid': '[1]', 'poly': '[1]', 'linear': '[1]', 'rbf': '[1]'}

Check the support vectors under different kernels...

{'sigmoid': array([[ 0.,  0.],
       [ -2.,  0.],
       [  1.,  1.],
       [ 10.,  1.]]), 'poly': array([[ 0.,  0.],
       [ 1.,  1.]]), 'linear': array([[ 0.,  0.],
       [ 1.,  1.]]), 'rbf': array([[  0.,   0.],
       [ -2.,   0.],
       [  1.,   1.],
       [ 10.,   1.]])}
```

Given a dataset, how can I split it into training and test data?

This is related to our project 1 problem. You need to partition the sample data and pick your training and test data such that 80% training; 20% test

Given a dataset, how can I split it into training and test data?

Use train_test_split module

from sklearn.model_selection **import** train_test_split

It has some reported issue for this input in PyCharm (interpreter: Anaconda 3.4) : it can't find this module!

To fix this, you need, install scikit-learn 'again' fro console

conda install scikit-learn

Or pip install scikit-learn

Given a dataset, how can I split it into training and test data?

Suppose your dataset is (data, label)→ response variables/labels
You want to have 80% training and 20% test

Set: test_size: 0.20
random_state marks the 'random process' in selecting training and test

```
training_data, test_data, training_data_label, test_data_label = train_test_split(data, label, test_size=0.20, random_state=42)
```

```
import numpy as np
import time

from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn import svm

iris  = datasets.load_iris()
data  = iris.data
label = iris.target


print(" data dimension:" + str((data.shape)))


test_percent = 0.3
training_data, test_data, training_data_label, test_data_label =
train_test_split(data,
   label, test_size=test_percent, random_state=42)

print("\n training_data size:{:d}".format(len(training_data)))
time.sleep(1)
```

```
N=7
print("\n The first ** {:d} **".format(N) + " training samples and its labels\n")
print(str(training_data[0:N])+"\n")
print(str(training_data_label[0:N]))

## build SVM learning machine

svm_learning_machine = svm.SVC(kernel='rbf', tol=0.0001, gamma=0.5, C=1)
svm_learning_machine.fit(training_data, training_data_label)

## prediction
predicted_test_data_label = svm_learning_machine.predict(test_data)

print("\n Predicted test data label vs. true test data label\n")
print(predicted_test_data_label)
print(test_data_label)
```

```
N=7
print("\n The first ** {:d} **".format(N) + " training samples and its labels\n")
print(str(training_data[0:N])+"\n")
print(str(training_data_label[0:N]))

## build SVM learning machine

svm_learning_machine = svm.SVC(kernel='rbf', tol=0.0001, gamma=0.5, C=1)
svm_learning_machine.fit(training_data, training_data_label)

## prediction
predicted_test_data_label = svm_learning_machine.predict(test_data)

print("\n Predicted test data label vs. true test data label\n")
print(predicted_test_data_label)
print(test_data_label)
```

```
Predicted test data label vs. true test data label

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

Check learning accuracy

```
learning_accuracy = svm_learning_machine.score(test_data, test_data_label)

print("SVM learning_accuracy is {:3.2f}".format(learning_accuracy)+"\n")

time.sleep(1)
```

Classification measures: accuracy, sensitivity, specificity, negative predictive ratios (NPR), and positive predicative ratios (PPR). (we assume a binary classification case)

Classification measures: accuracy, sensitivity, specificity, negative predictive ratios (NPR), and positive predicative ratios (PPR). (we assume a binary classification case)

TP (TN) is the number of positive (negative) samples correctly diagnosed, and FN (FP) is the number of positive (negative) samples incorrectly diagnosed.

Classification measures: accuracy, sensitivity, specificity, negative predictive ratios (NPR), and positive predicative ratios (PPR). (we assume a binary classification case)

TP (TN) is the number of positive (negative) samples correctly diagnosed, and FN (FP) is the number of positive (negative) samples incorrectly diagnosed.

$accuracy = \frac{TP+TN}{TP+FP+TN+FN}$ the percentage of correctly predicted subjects among all.

$sensitivity = \frac{TP}{TP+FN}$ the percentage of positive subjects correctly predicted

$specificity = \frac{TN}{TN+FP}$ the percentage of negative subjects correctly predicted

$NPR = \frac{TN}{TN+FN}$ negative subjects predictive ratio

$PPR = \frac{TP}{TP+FP}$ positive subjects predictive ratio

These measures are essential for checking overfitting and underfitting

These measures are essential for checking overfitting and underfitting

Overfitting: models are only too good for few data.

① Kernel matrix will be an identity matrix or near-identity-matrix
② Imbalanced sensitivity and specificity though accuracy may look fine.

Underfiting: models are too bad for this data

① Kernel matrix will be a matrix with all '1' entries
② Less than 50% accuracy

Overfitting can happen to any kernels, but mostly for nonlinear kernels instead of linear kernels

Under-fitting can happen to nonlinear kernel generally

Code these measures by yourselves

Write a package called compute_measure.py that has a function called compute_measure to compute these Classification measures

```
import numpy as np
import compute_measure

predicted_label = np.array( [-1, 1,  1,  -1,  1,  -1,  1,  -1,  -1,  -1])
true_label      = np.array( [ 1, 1,  1,  -1,  1,  1,  1,  1,   1,   1 ])

ans = compute_measure.compute_measure(predicted_label, true_label)

print("\n check the following classiciation measures: accuracy, sen, spec, ppr, npr\n")
print("{}".format(ans))
```

```
import numpy as np
import math

def compute_measure(predicted_label, true_label):
    t_idx = (predicted_label == true_label) # truely predicted
    f_idx = np.logical_not(t_idx)          # falsely predicted

    p_idx = (true_label > 0)               # positive targets
    n_idx = np.logical_not(p_idx)          # negative targets


    tp = np.sum( np.logical_and(t_idx,  p_idx))          # TP
    tn = np.sum( np.logical_and(t_idx, n_idx))           # TN

    # false positive: original negative but classified as positive
    # false negative: original positive but classified as negative

    fp = np.sum(n_idx) - tn
    fn = np.sum(p_idx) - tp
```

This is an almost pure python coding. You can use other packages to make this concise

```
    tp_fp_tn_fn_list=[]
    tp_fp_tn_fn_list.append(tp)
    tp_fp_tn_fn_list.append(fp)
    tp_fp_tn_fn_list.append(tn)
    tp_fp_tn_fn_list.append(fn)
    tp_fp_tn_fn_list=np.array(tp_fp_tn_fn_list)

    tp=tp_fp_tn_fn_list[0]
    fp=tp_fp_tn_fn_list[1]
    tn=tp_fp_tn_fn_list[2]
    fn=tp_fp_tn_fn_list[3]

 with np.errstate(divide='ignore'):
        sen = (1.0*tp)/(tp+fn)

    with np.errstate(divide='ignore'):
        spc = (1.0*tn)/(tn+fp)

    with np.errstate(divide='ignore'):
        ppr = (1.0*tp)/(tp+fp)
    with np.errstate(divide='ignore'):
        npr = (1.0*tn)/(tn+fn)
```

```
acc = (tp+tn)*1.0/(tp+fp+tn+fn)
ans=[]
ans.append(acc)
ans.append(sen)
ans.append(spc)
ans.append(ppr)
ans.append(npr)

return ans
```

Cross-validation methods

1. hold-out cross-validation
2. K-fold cross-validation
3. Leave-one-out cross-validation (LOOCV)

We need do regression in our implied volatility prediction, though classification methods can be used to predict stock future movement

The important measure we use is MSE

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(IV_i - predictedIV_i)^2$$

Demo: how to use k-NN to predict implied volatility

```
Input_train, Input_test, Response_train, Response_test = \
    train_test_split(Data_input, Data_response, test_size=0.2, random_state=42)


## Mean square error: MSE
def get_MSE(Error):
    mse=np.sum(np.power(Error,2))
    mse=mse/len(Error)
    return mse


# Train the model via KNN regression
# k=5 for training
kNN = KNeighborsRegressor(n_neighbors=5, weights='distance')
kNN.fit(Input_train, Response_train)
```

```
## performance analysis parameters

Error_KNN   = [None] * len(Input_test)
predictedIV = Error_KNN

##predicted implied votatility
predictedIV = kNN.predict(Input_test)
Error_KNN   = abs(Response_test - predictedIV)


# Model Evaluation

print('\n\nThe KNN Model Peformance Summary as follows:')
print('The MSE is          {:20.16f}'.format(get_MSE(Error_KNN)))
print('The mean error is    {:20.16f}'.format(np.mean(Error_KNN)))
print('The maximum error is {:20.16f}'.format(max(Error_KNN)))
print('The minimum error is {:20.16f}'.format(min(Error_KNN)))
```

SVR is the corresponding regression method for SVM. It has the same parameter setting as SVC

```
sklearn.svm.SVR(kernel='rbf', degree=3, gamma='auto', coef0=0.0, tol=0.001, C=1.0,
epsilon=0.1, shrinking=True, cache_size=200, verbose=False, max_iter=-1)
```

Use SVM to predict implied volatility and compare its performance with kNN