ModelSelection

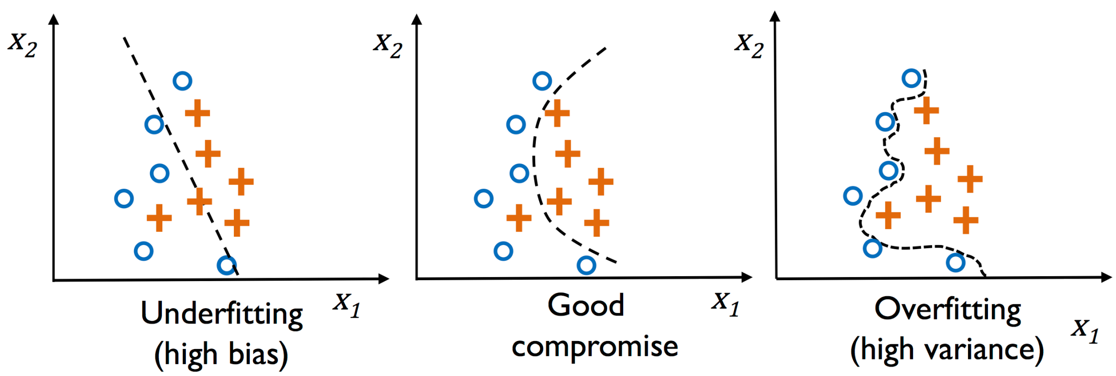
January 1, 2025

1 Model Comparison and Selection

- since no single classifier will work best for all the problems, we need to experiment with a handful
- need to effectively compare the models and select the best one for the problem

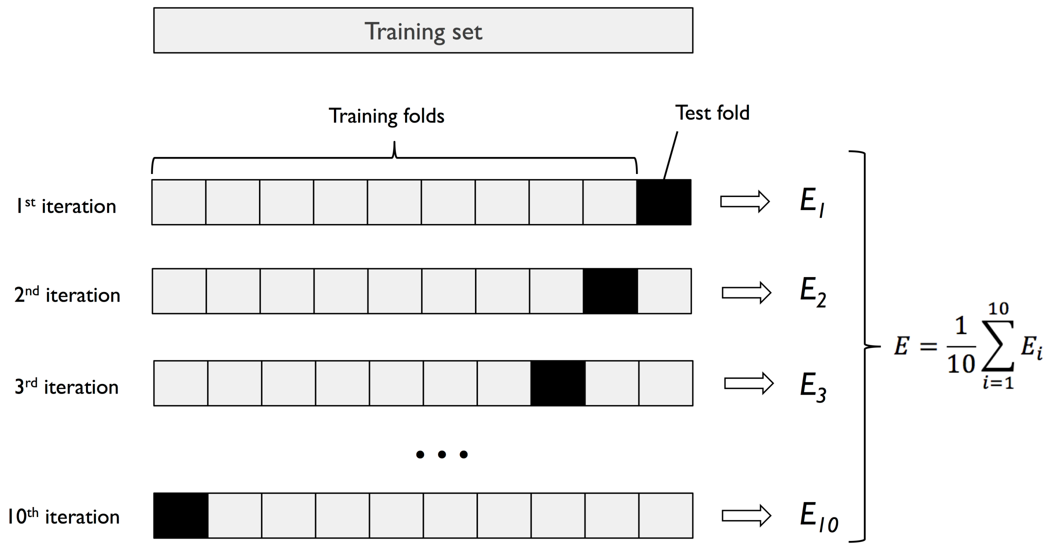
1.1 Over and under fitting models

- over or under fitting can occur if training data is not properly sampled or features are not properly selected
- models can suffer from underfitting (**high bias**) if the model is too simple
  - **bias** measures how far off the predictions are from the correct values in general if we rebuild the model multiple times on different datasets
- models can suffer from overfitting the training data (**high variance**) if the model is too complex for the underlying training data
  - **variance** measures the consistency (or variability) of the model prediction for classifying a particular example if we retrain the model multiple times, e.g., on different subsets of the training dataset
- the following figure demonstrates under and over fitting the models based



1.2 K-fold cross-validation

- k-fold cross-validation can help us obtain reliable estimates of the model's performance on unseen data



- stratified k-fold cross-validation can yield better bias and variance estimates, especially in cases of unequal class proportions
- https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html

1.2.1 Breast Cancer Wisconsin dataset

- details: https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)
- let's use the binary classification dataset for detecting breast cancer

[1]:
```
import pandas as pd
import numpy as np
```

[2]:
```
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/
↪breast-cancer-wisconsin/wdbc.data'
df = pd.read_csv(url, header=None)
```

[3]:
```
df
# Note col 0 is ID of the sample and col 1 is the corresponding diagnoses (M =␣
↪malignant, B = benign)
```

[3]:
```
            0  1      2      3       4       5        6        7        8  \
0      842302  M  17.99  10.38  122.80  1001.0  0.11840  0.27760  0.30010
1      842517  M  20.57  17.77  132.90  1326.0  0.08474  0.07864  0.08690
2    84300903  M  19.69  21.25  130.00  1203.0  0.10960  0.15990  0.19740
3    84348301  M  11.42  20.38   77.58   386.1  0.14250  0.28390  0.24140
4    84358402  M  20.29  14.34  135.10  1297.0  0.10030  0.13280  0.19800
..        … ..    …      …       …       …        …        …        …
564    926424  M  21.56  22.39  142.00  1479.0  0.11100  0.11590  0.24390
565    926682  M  20.13  28.25  131.20  1261.0  0.09780  0.10340  0.14400
566    926954  M  16.60  28.08  108.30   858.1  0.08455  0.10230  0.09251
```

```
567    927241  M  20.60  29.33  140.10  1265.0  0.11780  0.27700  0.35140
568     92751  B   7.76  24.54   47.92   181.0  0.05263  0.04362  0.00000

          9  …      22     23      24      25       26       27      28  \
0    0.14710  …  25.380  17.33  184.60  2019.0  0.16220  0.66560  0.7119
1    0.07017  …  24.990  23.41  158.80  1956.0  0.12380  0.18660  0.2416
2    0.12790  …  23.570  25.53  152.50  1709.0  0.14440  0.42450  0.4504
3    0.10520  …  14.910  26.50   98.87   567.7  0.20980  0.86630  0.6869
4    0.10430  …  22.540  16.67  152.20  1575.0  0.13740  0.20500  0.4000
..       …  …      …      …       …       …        …       …
564  0.13890  …  25.450  26.40  166.10  2027.0  0.14100  0.21130  0.4107
565  0.09791  …  23.690  38.25  155.00  1731.0  0.11660  0.19220  0.3215
566  0.05302  …  18.980  34.12  126.70  1124.0  0.11390  0.30940  0.3403
567  0.15200  …  25.740  39.42  184.60  1821.0  0.16500  0.86810  0.9387
568  0.00000  …   9.456  30.37   59.16   268.6  0.08996  0.06444  0.0000

         29      30       31
0    0.2654  0.4601  0.11890
1    0.1860  0.2750  0.08902
2    0.2430  0.3613  0.08758
3    0.2575  0.6638  0.17300
4    0.1625  0.2364  0.07678
..      …       …       …
564  0.2216  0.2060  0.07115
565  0.1628  0.2572  0.06637
566  0.1418  0.2218  0.07820
567  0.2650  0.4087  0.12400
568  0.0000  0.2871  0.07039

[569 rows x 32 columns]
```

[4]:
```
df.describe()
```

[4]:
```
                  0           2           3           4            5  \
count  5.690000e+02  569.000000  569.000000  569.000000   569.000000
mean   3.037183e+07   14.127292   19.289649   91.969033   654.889104
std    1.250206e+08    3.524049    4.301036   24.298981   351.914129
min    8.670000e+03    6.981000    9.710000   43.790000   143.500000
25%    8.692180e+05   11.700000   16.170000   75.170000   420.300000
50%    9.060240e+05   13.370000   18.840000   86.240000   551.100000
75%    8.813129e+06   15.780000   21.800000  104.100000   782.700000
max    9.113205e+08   28.110000   39.280000  188.500000  2501.000000

                6           7           8           9          10  …  \
count  569.000000  569.000000  569.000000  569.000000  569.000000  …
mean     0.096360    0.104341    0.088799    0.048919    0.181162  …
std      0.014064    0.052813    0.079720    0.038803    0.027414  …
```

```
min      0.052630    0.019380    0.000000    0.000000    0.106000  …
25%      0.086370    0.064920    0.029560    0.020310    0.161900  …
50%      0.095870    0.092630    0.061540    0.033500    0.179200  …
75%      0.105300    0.130400    0.130700    0.074000    0.195700  …
max      0.163400    0.345400    0.426800    0.201200    0.304000  …

               22          23          24           25          26  \
count  569.000000  569.000000  569.000000   569.000000  569.000000
mean    16.269190   25.677223  107.261213   880.583128    0.132369
std      4.833242    6.146258   33.602542   569.356993    0.022832
min      7.930000   12.020000   50.410000   185.200000    0.071170
25%     13.010000   21.080000   84.110000   515.300000    0.116600
50%     14.970000   25.410000   97.660000   686.500000    0.131300
75%     18.790000   29.720000  125.400000  1084.000000    0.146000
max     36.040000   49.540000  251.200000  4254.000000    0.222600

               27          28          29          30          31
count  569.000000  569.000000  569.000000  569.000000  569.000000
mean     0.254265    0.272188    0.114606    0.290076    0.083946
std      0.157336    0.208624    0.065732    0.061867    0.018061
min      0.027290    0.000000    0.000000    0.156500    0.055040
25%      0.147200    0.114500    0.064930    0.250400    0.071460
50%      0.211900    0.226700    0.099930    0.282200    0.080040
75%      0.339100    0.382900    0.161400    0.317900    0.092080
max      1.058000    1.252000    0.291000    0.663800    0.207500

[8 rows x 31 columns]
```

[5]:
```python
# Let's create X and y numpy ndarrays
X = df.loc[:, 2:].values
y = df.loc[:, 1].values
```

[6]:
```python
y
```

[6]:
```
array(['M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M',
       'M', 'M', 'M', 'M', 'M', 'M', 'B', 'B', 'B', 'M', 'M', 'M', 'M',
       'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'B', 'M',
       'M', 'M', 'M', 'M', 'M', 'M', 'M', 'B', 'M', 'B', 'B', 'B', 'B',
       'B', 'M', 'M', 'B', 'M', 'M', 'B', 'B', 'B', 'B', 'M', 'B', 'M',
       'M', 'B', 'B', 'B', 'B', 'M', 'B', 'M', 'M', 'B', 'M', 'B', 'M',
       'M', 'B', 'B', 'B', 'M', 'M', 'B', 'M', 'M', 'M', 'B', 'B', 'B',
       'M', 'B', 'B', 'M', 'M', 'B', 'B', 'B', 'M', 'M', 'B', 'B', 'B',
       'B', 'M', 'B', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
       'M', 'M', 'M', 'B', 'M', 'M', 'B', 'B', 'B', 'M', 'M', 'B', 'M',
       'B', 'M', 'M', 'B', 'M', 'M', 'B', 'B', 'M', 'B', 'B', 'M', 'B',
       'B', 'B', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
       'M', 'B', 'B', 'B', 'B', 'M', 'M', 'B', 'M', 'B', 'B', 'M', 'M',
```

```
'B', 'B', 'M', 'M', 'B', 'B', 'B', 'B', 'M', 'B', 'B', 'M', 'M',
'M', 'B', 'M', 'B', 'M', 'B', 'B', 'B', 'M', 'B', 'B', 'M', 'M',
'B', 'M', 'M', 'M', 'M', 'B', 'M', 'M', 'M', 'B', 'M', 'B', 'M',
'B', 'B', 'M', 'B', 'M', 'M', 'M', 'M', 'B', 'B', 'M', 'M', 'B',
'B', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'M', 'M', 'B', 'B', 'M',
'B', 'B', 'M', 'M', 'B', 'M', 'B', 'B', 'B', 'B', 'M', 'B', 'B',
'B', 'B', 'B', 'M', 'B', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M',
'M', 'M', 'M', 'M', 'M', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'M',
'B', 'M', 'B', 'B', 'M', 'B', 'B', 'M', 'B', 'M', 'M', 'B', 'B',
'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'M', 'B',
'B', 'M', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
'B', 'B', 'B', 'B', 'B', 'M', 'B', 'B', 'B', 'M', 'B', 'M', 'B',
'B', 'B', 'B', 'M', 'M', 'M', 'B', 'B', 'B', 'B', 'M', 'B', 'M',
'B', 'M', 'B', 'B', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
'M', 'M', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
'B', 'M', 'M', 'B', 'M', 'M', 'M', 'B', 'M', 'M', 'B', 'B', 'B',
'B', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'M', 'B', 'B', 'B', 'M',
'B', 'B', 'M', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'M', 'B', 'B',
'B', 'B', 'B', 'B', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'M', 'B',
'B', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
'B', 'M', 'B', 'M', 'M', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'M',
'B', 'B', 'M', 'B', 'M', 'B', 'B', 'M', 'B', 'M', 'B', 'B', 'B',
'B', 'B', 'B', 'B', 'B', 'M', 'M', 'B', 'B', 'B', 'B', 'B', 'B',
'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'M', 'B',
'B', 'B', 'B', 'B', 'B', 'B', 'M', 'B', 'M', 'B', 'B', 'M', 'B',
'B', 'B', 'B', 'B', 'M', 'M', 'B', 'M', 'B', 'M', 'B', 'B', 'B',
'B', 'B', 'M', 'B', 'B', 'M', 'B', 'M', 'B', 'M', 'M', 'B', 'B',
'B', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
'M', 'B', 'M', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
'B', 'B', 'B', 'M', 'M', 'M', 'M', 'M', 'M', 'B'], dtype=object)
```

[7]:
```python
# let's encode the labels with LabelEncoder
from sklearn.preprocessing import LabelEncoder
```

[8]:
```python
le = LabelEncoder()
y = le.fit_transform(y)
```

[9]:
```python
y[:10]
```

[9]:
```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

[10]:
```python
y.shape
```

[10]:
```
(569,)
```

[11]:
```python
X.shape
```

[11]: (569, 30)

[12]:
```
le.classes_
# 0 is Benign (Not-Cancer) and 1 is Malignant (Cancer)
```

[12]: array(['B', 'M'], dtype=object)

[13]:
```
# let's Scale the data using StandardScaler
from sklearn.preprocessing import StandardScaler
```

[14]:
```
sc = StandardScaler()
sc.fit(X) # fit the whole data to calculate mean and standard deviation
X_sc = sc.transform(X) # transform training set
```

[15]:
```
# let's do the StratifiedKFold cross validation
from sklearn.model_selection import StratifiedKFold
# use logistic regression classifier
from sklearn.linear_model import LogisticRegression
#https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.
 ↪LogisticRegression.html
```

[16]:
```
kfold = StratifiedKFold(n_splits=10)
```

[17]:
```
scores = []
for k, (train, test) in enumerate(kfold.split(X_sc, y)): # iterator
    lr_model = LogisticRegression(random_state=1, solver='lbfgs')
    #print(train.shape, test.shape)
    lr_model.fit(X_sc[train], y[train])
    score = lr_model.score(X_sc[test], y[test])
    scores.append(score)
    print(f'Fold:{k+1:2d}, Class dist.:{np.bincount(y[train])}, Acc: {score:.
 ↪3f}')
```

```
Fold: 1, Class dist.:[322 190], Acc: 0.982
Fold: 2, Class dist.:[322 190], Acc: 0.982
Fold: 3, Class dist.:[321 191], Acc: 0.982
Fold: 4, Class dist.:[321 191], Acc: 0.965
Fold: 5, Class dist.:[321 191], Acc: 0.982
Fold: 6, Class dist.:[321 191], Acc: 0.982
Fold: 7, Class dist.:[321 191], Acc: 0.947
Fold: 8, Class dist.:[321 191], Acc: 1.000
Fold: 9, Class dist.:[321 191], Acc: 1.000
Fold:10, Class dist.:[322 191], Acc: 0.982
```

[18]:
```
print(f'CV accuracy : {np.mean(scores):.3f}, +/- {np.std(scores):.3f}')
```

```
CV accuracy : 0.981, +/- 0.015
```

[19]:
```python
# better: use scikit learn's cross_val_score
from sklearn.model_selection import cross_val_score
```

[20]:
```python
lr_model = LogisticRegression(random_state=1, solver='lbfgs')
scores = cross_val_score(estimator=lr_model, X=X_sc, y=y, cv=10, n_jobs=-1)
# n_jobs = -1 means use all available processors to do computation in parallel
```

[21]:
```python
print(f'CV accuracy scores: {scores}')
```

```
CV accuracy scores: [0.98245614 0.98245614 0.98245614 0.96491228 0.98245614
0.98245614
 0.94736842 1.         1.         0.98214286]
```

[22]:
```python
print(f'CV accuracy: {np.mean(scores):.3f}, +/- {np.std(scores):.3f}')
```

```
CV accuracy: 0.981, +/- 0.015
```

[23]:
```python
# let's compare a handful of Classifiers
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
```

[24]:
```python
names = ["KNN", "Linear SVM", "RBF SVM", "Gaussian Process",
         "Decision Tree", "Random Forest", "Neural Net", "AdaBoost",
         "Naive Bayes", "QDA", 'Logistic Reg']
scores = [] # store (name, mean, std_dev) for each classifier
classifiers = [
    KNeighborsClassifier(3),
    SVC(kernel="linear", C=0.025),
    SVC(gamma=2, C=1),
    GaussianProcessClassifier(1.0 * RBF(1.0)),
    DecisionTreeClassifier(max_depth=5),
    RandomForestClassifier(max_depth=5, n_estimators=10, max_features=1),
    MLPClassifier(alpha=1, max_iter=1000),
    AdaBoostClassifier(),
    GaussianNB(),
    QuadraticDiscriminantAnalysis(),
    LogisticRegression(random_state=1, solver='lbfgs')
]

# iterate over classifiers
for name, clf in zip(names, classifiers):
```

```
    cvs = cross_val_score(estimator=clf, X=X_sc, y=y, cv=10, n_jobs=-1)
    scores.append((name, np.mean(cvs), np.std(cvs)))
```

```
/usr/local/lib/python3.12/site-
packages/sklearn/ensemble/_weight_boosting.py:519: FutureWarning: The SAMME.R
algorithm (the default) is deprecated and will be removed in 1.6. Use the SAMME
algorithm to circumvent this warning.
  warnings.warn(
/usr/local/lib/python3.12/site-
packages/sklearn/ensemble/_weight_boosting.py:519: FutureWarning: The SAMME.R
algorithm (the default) is deprecated and will be removed in 1.6. Use the SAMME
algorithm to circumvent this warning.
  warnings.warn(
/usr/local/lib/python3.12/site-
packages/sklearn/ensemble/_weight_boosting.py:519: FutureWarning: The SAMME.R
algorithm (the default) is deprecated and will be removed in 1.6. Use the SAMME
algorithm to circumvent this warning.
  warnings.warn(
/usr/local/lib/python3.12/site-
packages/sklearn/ensemble/_weight_boosting.py:519: FutureWarning: The SAMME.R
algorithm (the default) is deprecated and will be removed in 1.6. Use the SAMME
algorithm to circumvent this warning.
  warnings.warn(
/usr/local/lib/python3.12/site-
packages/sklearn/ensemble/_weight_boosting.py:519: FutureWarning: The SAMME.R
algorithm (the default) is deprecated and will be removed in 1.6. Use the SAMME
algorithm to circumvent this warning.
  warnings.warn(
/usr/local/lib/python3.12/site-
packages/sklearn/ensemble/_weight_boosting.py:519: FutureWarning: The SAMME.R
algorithm (the default) is deprecated and will be removed in 1.6. Use the SAMME
algorithm to circumvent this warning.
  warnings.warn(
/usr/local/lib/python3.12/site-
packages/sklearn/ensemble/_weight_boosting.py:519: FutureWarning: The SAMME.R
algorithm (the default) is deprecated and will be removed in 1.6. Use the SAMME
algorithm to circumvent this warning.
  warnings.warn(
/usr/local/lib/python3.12/site-
packages/sklearn/ensemble/_weight_boosting.py:519: FutureWarning: The SAMME.R
algorithm (the default) is deprecated and will be removed in 1.6. Use the SAMME
algorithm to circumvent this warning.
  warnings.warn(
/usr/local/lib/python3.12/site-
packages/sklearn/ensemble/_weight_boosting.py:519: FutureWarning: The SAMME.R
algorithm (the default) is deprecated and will be removed in 1.6. Use the SAMME
algorithm to circumvent this warning.
```

```
  warnings.warn(
/usr/local/lib/python3.12/site-
packages/sklearn/ensemble/_weight_boosting.py:519: FutureWarning: The SAMME.R
algorithm (the default) is deprecated and will be removed in 1.6. Use the SAMME
algorithm to circumvent this warning.
  warnings.warn(
```

[25]:
```python
scores
```

[25]:
```
[('KNN', 0.9647869674185465, 0.02239183921884522),
 ('Linear SVM', 0.9736215538847116, 0.021152440486425998),
 ('RBF SVM', 0.6274122807017544, 0.006965956216784447),
 ('Gaussian Process', 0.9789473684210526, 0.017189401703741607),
 ('Decision Tree', 0.9175438596491228, 0.04004460424741518),
 ('Random Forest', 0.9403195488721805, 0.03343225060091806),
 ('Neural Net', 0.975407268170426, 0.02104239610504222),
 ('AdaBoost', 0.963095238095238, 0.027680410820357663),
 ('Naive Bayes', 0.9315162907268169, 0.0327113878182645),
 ('QDA', 0.9560776942355889, 0.02110040899790857),
 ('Logistic Reg', 0.9806704260651629, 0.01456955548732776)]
```

[26]:
```python
# let's sort the scores in descending order of accuracy
scores.sort(key=lambda t: t[1], reverse=True)
```

[27]:
```python
scores
```

[27]:
```
[('Logistic Reg', 0.9806704260651629, 0.01456955548732776),
 ('Gaussian Process', 0.9789473684210526, 0.017189401703741607),
 ('Neural Net', 0.975407268170426, 0.02104239610504222),
 ('Linear SVM', 0.9736215538847116, 0.021152440486425998),
 ('KNN', 0.9647869674185465, 0.02239183921884522),
 ('AdaBoost', 0.963095238095238, 0.027680410820357663),
 ('QDA', 0.9560776942355889, 0.02110040899790857),
 ('Random Forest', 0.9403195488721805, 0.03343225060091806),
 ('Naive Bayes', 0.9315162907268169, 0.0327113878182645),
 ('Decision Tree', 0.9175438596491228, 0.04004460424741518),
 ('RBF SVM', 0.6274122807017544, 0.006965956216784447)]
```

1.3 ROC curve

- Receiver Operating Characteristic (ROC) graphs are used to select models for classification based on the performance with respect to the FPR and TPR
- the diagonal of the ROC curve can be interpreted as *random guessing*
  - classification models that fall below the diagonal are considered as worse than random guessing
- a perfect classifier would fall into the top-left corner of the graph with a **TPR of 1** and and an **FPR of 0**

- based on ROC curve, we can compute ROC area under the curve (ROC AUC) to characterize the performance of a classification model

1.3.1 Logistic Regression - ROC curve for cross validation

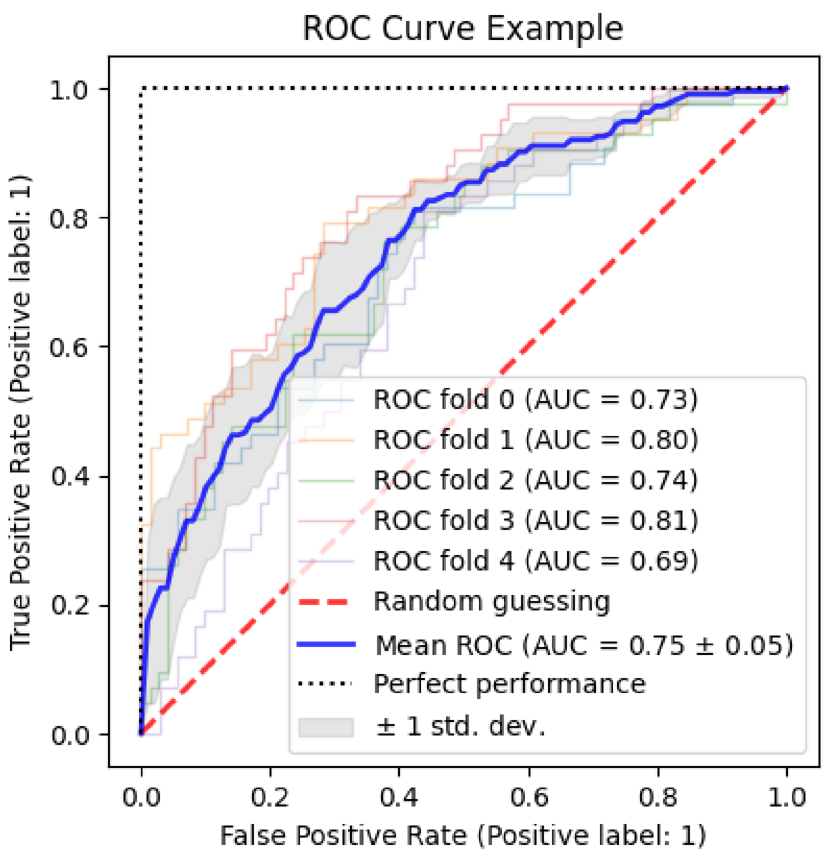
- https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc_crossval.html

[30]:
```
from sklearn.metrics import RocCurveDisplay, auc
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import StratifiedKFold
```

[31]:
```
# To generate more representitive ROC graph,
# we'll use just 2 features 4 and 14 making it more challenging for the␣
↪classifier
X_train = X_sc[:, [4, 14]]
```

[35]:
```
cv = StratifiedKFold(n_splits=5) # just to 5 fold
classifier = LogisticRegression(random_state=1, solver='lbfgs')
tprs = []
aucs = []
mean_fpr = np.linspace(0, 1, 100)
fig, ax = plt.subplots()

# create and add ROC for each  fold
for i, (train, test) in enumerate(cv.split(X_train, y)): # iterator
    classifier.fit(X_train[train], y[train])
    viz = RocCurveDisplay.from_estimator(classifier, X_train[test], y[test],
                         name=f'ROC fold {i}',
                         alpha=0.3, lw=1, ax=ax)
    interp_tpr = np.interp(mean_fpr, viz.fpr, viz.tpr)
    interp_tpr[0] = 0.0
    tprs.append(interp_tpr)
    aucs.append(viz.roc_auc)

# add curve for random guessing
ax.plot([0, 1], [0, 1], linestyle='--', lw=2, color='r',
        label='Random guessing', alpha=.8)

mean_tpr = np.mean(tprs, axis=0)
mean_tpr[-1] = 1.0
mean_auc = auc(mean_fpr, mean_tpr)
std_auc = np.std(aucs)

# add curve for mean scores
ax.plot(mean_fpr, mean_tpr, color='b',
        label=r'Mean ROC (AUC = %0.2f $\pm$ %0.2f)' % (mean_auc, std_auc),
        lw=2, alpha=.8)
```

```python
# add curve for a perfect score
ax.plot([0, 0, 1],
        [0, 1, 1], linestyle=':', color='black', label='Perfect performance')

std_tpr = np.std(tprs, axis=0)
tprs_upper = np.minimum(mean_tpr + std_tpr, 1)
tprs_lower = np.maximum(mean_tpr - std_tpr, 0)
ax.fill_between(mean_fpr, tprs_lower, tprs_upper, color='grey', alpha=.2,
                label=r'$\pm$ 1 std. dev.')

ax.set(xlim=[-0.05, 1.05], ylim=[-0.05, 1.05],
       title="ROC Curve Example")
ax.legend(loc="lower right")
plt.show()
```



1.4 ROC Curve to compare models

[37]:
```
import pandas as pd
import numpy as np
from sklearn.metrics import roc_curve, auc, roc_auc_score
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from itertools import cycle
from sklearn.model_selection import train_test_split
```

[38]:
```
# let's compare a handful of Classifiers
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
```

[39]:
```
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/
 ↪breast-cancer-wisconsin/wdbc.data'
df = pd.read_csv(url, header=None)
```

[40]:
```
X = df.loc[:, 2:].values
y = df.loc[:, 1].values
```

[41]:
```
le = LabelEncoder()
y = le.fit_transform(y)
```

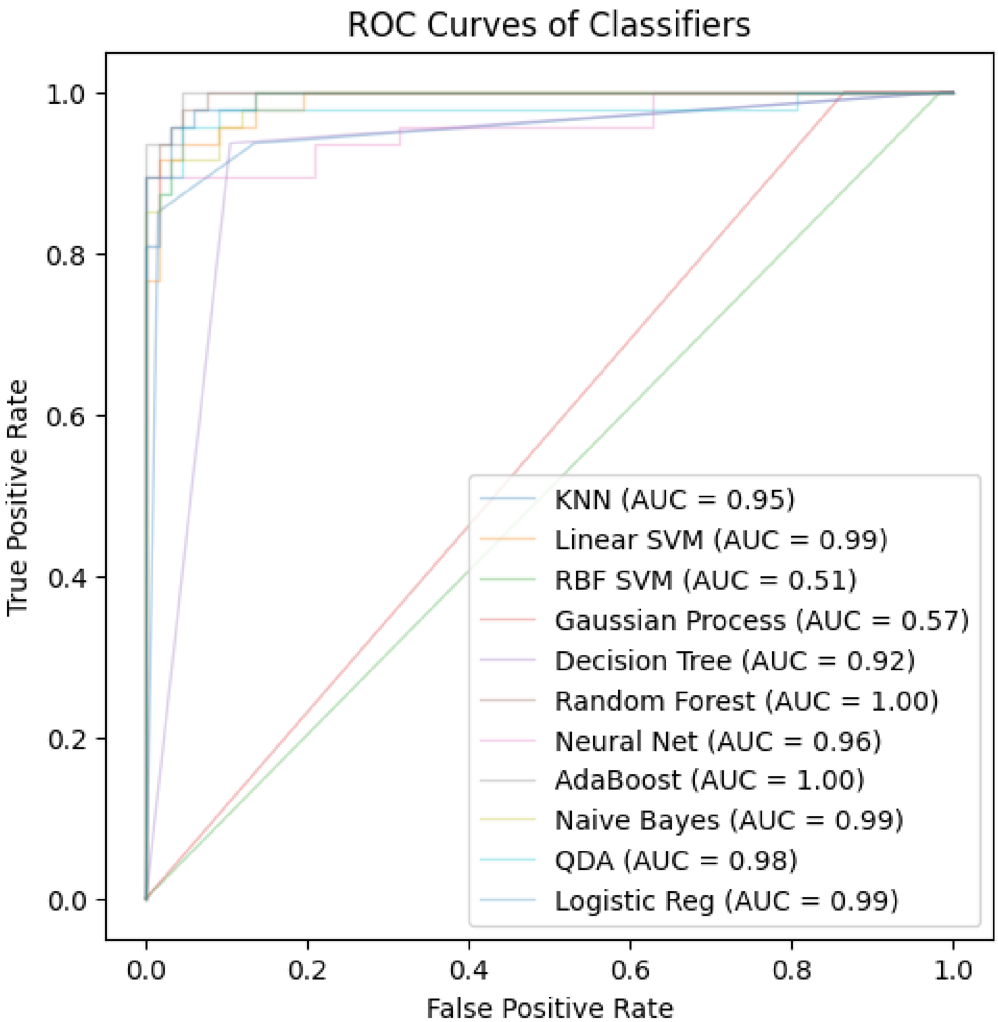
[42]:
```
sc = StandardScaler()
sc.fit(X) # fit the whole data to calculate mean and standard deviation
X_sc = sc.transform(X) # transform training set
```

[43]:
```
names = ["KNN", "Linear SVM", "RBF SVM", "Gaussian Process",
         "Decision Tree", "Random Forest", "Neural Net", "AdaBoost",
         "Naive Bayes", "QDA", 'Logistic Reg']

classifiers = [
    KNeighborsClassifier(2),
    SVC(kernel="linear", C=0.025),
    SVC(gamma=2, C=1),
    GaussianProcessClassifier(),
    DecisionTreeClassifier(),
    RandomForestClassifier(),
```

```
    MLPClassifier(),
    AdaBoostClassifier(),
    GaussianNB(),
    QuadraticDiscriminantAnalysis(),
    LogisticRegression(random_state=1, solver='lbfgs')
]
mean_fpr = np.linspace(0, 1, 100)
#cv = StratifiedKFold(n_splits=5) # just to 5 fold
```

[45]:
```
# let's plot the ROC Curves for all the classifiers
fig, ax = plt.subplots(figsize=(10, 6))
lw=2

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.2,
                                                    random_state=0)
for name, classifier in zip(names, classifiers):
    classifier.fit(X_train, y_train)
    RocCurveDisplay.from_estimator(classifier, X_test, y_test,
                         name=f'{name}',
                         alpha=0.3, lw=1, ax=ax)

ax.set(xlim=[-0.05, 1.05], ylim=[-0.05, 1.05], title="ROC Curve Example")
ax.legend(loc="lower right")
plt.title("ROC Curves of Classifiers")
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend(loc="lower right")
plt.show()
```

```
/usr/local/lib/python3.12/site-
packages/sklearn/ensemble/_weight_boosting.py:519: FutureWarning: The SAMME.R
algorithm (the default) is deprecated and will be removed in 1.6. Use the SAMME
algorithm to circumvent this warning.
  warnings.warn(
/usr/local/lib/python3.12/site-packages/sklearn/linear_model/_logistic.py:469:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  n_iter_i = _check_optimize_result(
```


ROC Curves of Classifiers
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
KNN (AUC = 0.95)
Linear SVM (AUC = 0.99)
RBF SVM (AUC = 0.51)
Gaussian Process (AUC = 0.57)
Decision Tree (AUC = 0.92)
Random Forest (AUC = 1.00)
Neural Net (AUC = 0.96)
AdaBoost (AUC = 1.00)
Naive Bayes (AUC = 0.99)
QDA (AUC = 0.98)
Logistic Reg (AUC = 0.99)

```
[ ]:
```