4_Evaluation_students

July 13, 2019

1 Hands-On Machine Learning

1.1 Session 4: Evaluation, Validation and Model Assessment

by Laxmi Gupta, Christoph Haarburger

1.1.1 Goals of this Session

In this session you will... * learn to assess a model's performance * get to know the most common evaluation measures * learn how to tune a model's hyperparameters

1.1.2 Iris Dataset

You already know the Iris dataset from you first session. Anyway, here's a quick recap: The dataset consists of three species (=classes) of the Iris plant (Setosa, Versicolor and Virginica). For each sample, four features have been measured (sepal length, sepal width, petal length, petal width).

The goal ist to train a model that can classify a given sample into one of the three species. Based on your knowledge of classifiers from the 1st session, today we'll focus more on the evaluation of performance and tuning of hyperparameters rather than the classifier itself.

```
In [1]: %matplotlib inline
```

```
In [2]: import seaborn as sns
        from evaluation import load_iris
        data = load_iris()
```

Before training any kind of model it is always a good idea to make ourselves familiar with the raw data and explore basic properties of the dataset. The dataset has already been loaded as a `pandas.DataFrame` which is a very handy data type for our exploration.

```
In [3]: data[:15]
```

```
Out[3]:     sepal_length  sepal_width  petal_length  petal_width species
        0            5.1          3.5           1.4          0.2  setosa
        1            4.9          3.0           1.4          0.2  setosa
        2            4.7          3.2           1.3          0.2  setosa
        3            4.6          3.1           1.5          0.2  setosa
        4            5.0          3.6           1.4          0.2  setosa
```

5	5.4	3.9	1.7	0.4	setosa
6	4.6	3.4	1.4	0.3	setosa
7	5.0	3.4	1.5	0.2	setosa
8	4.4	2.9	1.4	0.2	setosa
9	4.9	3.1	1.5	0.1	setosa
10	5.4	3.7	1.5	0.2	setosa
11	4.8	3.4	1.6	0.2	setosa
12	4.8	3.0	1.4	0.1	setosa
13	4.3	3.0	1.1	0.1	setosa
14	5.8	4.0	1.2	0.2	setosa

You my find it interesting to look at the raw numbers, but with our goal of building a classifier in mind we particularly want to explore statistical properties of the dataset.

```
In [4]: data.describe()
```

Out[4]:

	sepal_length	sepal_width	petal_length	petal_width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333
std	0.828066	0.435866	1.765298	0.762238
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

Task: Find out if the prevalence of the three classes is equal!

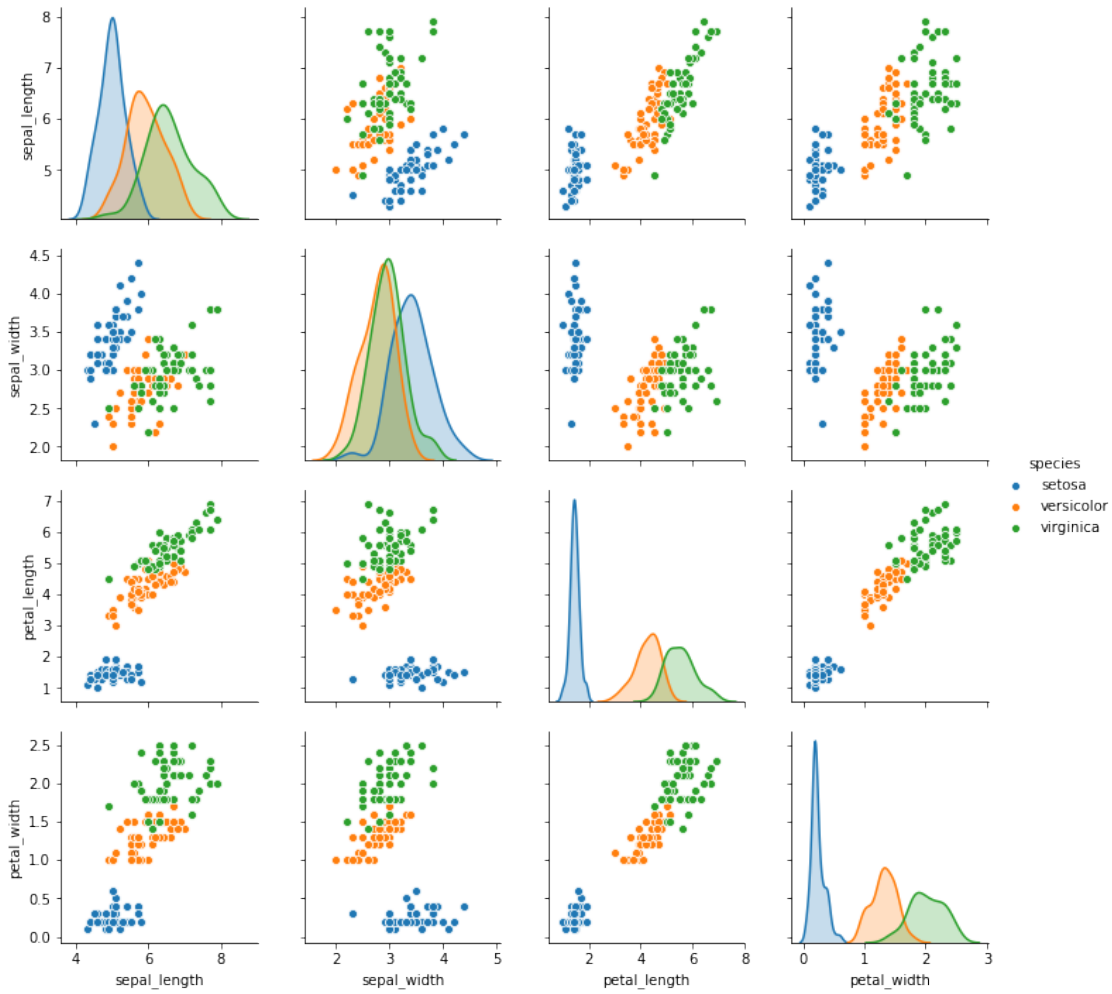
```
In [5]: import numpy as np
        y = data['species'].values # turn pandas.DataFrame into np.ndarray
        #print(y)
        unique, counts = np.unique(y, return_counts=True)
        print(counts)
        print(unique)
        #dict(zip(unique, counts))
```

```
[50 50 50]
['setosa' 'versicolor' 'virginica']
```

We're lucky, all classes are represented in equal ratios .

If the number of features is relatively small as in our case, it is also a good idea to plot features pairswise against each other and inspect the feature scores for all classes as histograms.

```
In [6]: _ = sns.pairplot(data, hue='species')
```



Task: Can the classification problem be solved by a linear classifier?

**Answer:

Now that we have gained some intuitution on the problem to be solved und the underlying data, let's get to what we are actually up to: Classification! As a first shot, it is always a good idea to utilize a classifier that is easy to understand and easy to interpret. K-Nearest-Neighbors does the job...

```
In [7]: from sklearn.neighbors import KNeighborsClassifier as KNN

        X = data[['sepal_length', 'sepal_width', 'petal_length', 'petal_width']].as_matrix() #

        model = KNN(n_neighbors=2)
        model.fit(X, y)
        y_pred = model.predict(X)
        print(y_pred)
```

```
['setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa'
 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa'
```

```
 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa'
 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa'
 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa'
 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa'
 'setosa' 'setosa' 'versicolor' 'versicolor' 'versicolor' 'versicolor'
 'versicolor' 'versicolor' 'versicolor' 'versicolor' 'versicolor'
 'versicolor' 'versicolor' 'versicolor' 'versicolor' 'versicolor'
 'versicolor' 'versicolor' 'versicolor' 'versicolor' 'versicolor'
 'versicolor' 'versicolor' 'versicolor' 'versicolor' 'versicolor'
 'versicolor' 'versicolor' 'versicolor' 'versicolor' 'versicolor'
 'versicolor' 'versicolor' 'versicolor' 'versicolor' 'versicolor'
 'versicolor' 'versicolor' 'versicolor' 'versicolor' 'versicolor'
 'versicolor' 'virginica' 'virginica' 'virginica' 'virginica' 'virginica'
 'virginica' 'versicolor' 'virginica' 'virginica' 'virginica' 'virginica'
 'virginica' 'virginica' 'virginica' 'virginica' 'virginica' 'virginica'
 'virginica' 'virginica' 'versicolor' 'virginica' 'virginica' 'virginica'
 'virginica' 'virginica' 'virginica' 'virginica' 'virginica' 'virginica'
 'virginica' 'virginica' 'virginica' 'virginica' 'versicolor' 'virginica'
 'virginica' 'virginica' 'virginica' 'virginica' 'virginica' 'virginica'
 'virginica' 'virginica' 'virginica' 'virginica' 'virginica' 'virginica'
 'virginica' 'virginica' 'virginica']


/usr/local/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:3: FutureWarning: Metho
  This is separate from the ipykernel package so we can avoid doing imports until
```

Task: To interpret the result, implement the accuracy measure and calculate the accuracy for our predictions. Accuracy is defined as $\frac{\text{true}_{\text{positive}}+\text{true}_{\text{negative}}}{\text{positive}+\text{negative}}$.

```
In [8]: def accuracy(y_true, y_pred):
            #acc_score=accuracy_score(y_true,y_pred)
            #print(a
            tp = 0
            tn = 0
            for i in range(len(y_true)):
                if(y_pred[i] == y_true[i]):
                    tp = tp + 1
                #else
                #tn = tn + 1

            accuracy_score = (tp + tn)/len(y_true)
            return accuracy_score

        print(y)
        print(y_pred)
```

```
        #print(accuracy_score(y, y_pred)),
        print(accuracy(y, y_pred))
```

```
['setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa'
 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa'
 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa'
 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa'
 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa'
 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa'
 'setosa' 'setosa' 'versicolor' 'versicolor' 'versicolor' 'versicolor'
 'versicolor' 'versicolor' 'versicolor' 'versicolor' 'versicolor'
 'versicolor' 'versicolor' 'versicolor' 'versicolor' 'versicolor'
 'versicolor' 'versicolor' 'versicolor' 'versicolor' 'versicolor'
 'versicolor' 'versicolor' 'versicolor' 'versicolor' 'versicolor'
 'versicolor' 'versicolor' 'versicolor' 'versicolor' 'versicolor'
 'versicolor' 'versicolor' 'versicolor' 'versicolor' 'versicolor'
 'versicolor' 'versicolor' 'versicolor' 'versicolor' 'versicolor'
 'versicolor' 'virginica' 'virginica' 'virginica' 'virginica' 'virginica'
 'virginica' 'virginica' 'virginica' 'virginica' 'virginica' 'virginica'
 'virginica' 'virginica' 'virginica' 'virginica' 'virginica' 'virginica'
 'virginica' 'virginica' 'virginica' 'virginica' 'virginica' 'virginica'
 'virginica' 'virginica' 'virginica' 'virginica' 'virginica' 'virginica'
 'virginica' 'virginica' 'virginica' 'virginica' 'virginica' 'virginica'
 'virginica' 'virginica' 'virginica' 'virginica' 'virginica' 'virginica'
 'virginica' 'virginica' 'virginica']
['setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa'
 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa'
 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa'
 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa'
 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa'
 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa' 'setosa'
 'setosa' 'setosa' 'versicolor' 'versicolor' 'versicolor' 'versicolor'
 'versicolor' 'versicolor' 'versicolor' 'versicolor' 'versicolor'
 'versicolor' 'versicolor' 'versicolor' 'versicolor' 'versicolor'
 'versicolor' 'versicolor' 'versicolor' 'versicolor' 'versicolor'
 'versicolor' 'versicolor' 'versicolor' 'versicolor' 'versicolor'
 'versicolor' 'versicolor' 'versicolor' 'versicolor' 'versicolor'
 'versicolor' 'versicolor' 'versicolor' 'versicolor' 'versicolor'
 'versicolor' 'versicolor' 'versicolor' 'versicolor' 'versicolor'
 'versicolor' 'virginica' 'virginica' 'virginica' 'virginica' 'virginica'
 'virginica' 'versicolor' 'virginica' 'virginica' 'virginica' 'virginica'
 'virginica' 'virginica' 'virginica' 'virginica' 'virginica' 'virginica'
 'virginica' 'virginica' 'versicolor' 'virginica' 'virginica' 'virginica'
```

```
 'virginica' 'virginica' 'virginica' 'virginica' 'virginica' 'virginica'
 'virginica' 'virginica' 'virginica' 'virginica' 'versicolor' 'virginica'
 'virginica' 'virginica' 'virginica' 'virginica' 'virginica' 'virginica'
 'virginica' 'virginica' 'virginica' 'virginica' 'virginica' 'virginica'
 'virginica' 'virginica' 'virginica']
0.98
```

Task: Why is the classification accuracy nearly perfect?
Answer:
Training set is used for testing also

1.2 Holdout Sets

A much better way to evaluate performance is to split all available data into a training and test set, where the training set is used for training the model's parameters and the test set is utilized for assessing performance.

Task: Implement the same pipeline as in the last code cell but with a separate training and test set (50/50)!

```
In [9]: from sklearn.model_selection import train_test_split

        #, y = np.arange(10).reshape((5, 2)), range(5)
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.50, random_state=
        #print(len(X_train))
        #print(len(X_test))
        #print(len(y_train))
        #print(len(y_test))
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        print(accuracy(y_test, y_pred))
```

```
0.96
```

That looks a lot more realistic!

Task: When using `train_test_split`, the samples are *randomly* drawn to one of the sets. What happens, if we use a different split? Determine the 95% confidence intervals for random 100 splits.
Hint: `np.percentile`

```
In [10]: acc_mat=[]
         for i in np.arange(1,100,1):
             X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=np.random.uni
             model.fit(X_train, y_train)
             y_pred = model.predict(X_test)
             acc_mat.append(accuracy(y_test, y_pred))


         #print(acc_mat)
```

```
        acc_mat.sort()
        #a = np.asarray(acc_mat)
        #print(np.percentile(a,2.5, axis = 0))
        #print(np.percentile(a,97.5, axis = 0))
        print(np.percentile(acc_mat, 2.5))
        print(np.percentile(acc_mat, 97.5))
```

```
0.856721470019
1.0
```

Task: What properties should a 'good' split have with respect to the dataset?
Answer:

1.3 Cross Validation

One disadvantage of the holdout set method is that half the dataset does not contribute to the training of the model. Especially if the initial set of training data is small this is suboptimal since statistical learning methods tend to perform worse when trained on fewer observations.

In cross validation we can overcome this by partitioning the data set in `k` "folds" of approximately equal size. The first fold is used as a test set and the model is trained on the remaining `k` - 1 folds. The test error is then co puted on the test set and the process is repeated `k` times, resulting in `k` estimates of the test error.

Task: Implement the `CrossValidation` class as sketched in the following cell! Hint: This is advanced task that is not mandatory to be solved.

Task: Use the `CrossValidation` class to evaluate the test error of the same KNN model as above but with 10 fold cross validation. What's the mean test error? If you did not solve the previous task on your own, you can use `KFold` from `scikit-learn`.

```
In [11]: #from sklearn.model_selection import KFold

        #kf = KFold(n_splits=10)

        from numpy import array
        from sklearn.model_selection import KFold
        kfold = KFold(10)
        # enumerate splits
        acc_mat=[]
        for train, test in kfold.split(X):
                #print('train: %s, test: %s' % (, X[test]))
            X_train=X[train]
            X_test =X[test]
            y_train=y[train]
            y_test =y[test]
            model = KNN(n_neighbors=2)
            model.fit(X_train, y_train)
            y_pred = model.predict(X_test)
```

```
            acc_mat.append(accuracy(y_test, y_pred))

        print(np.mean(acc_mat))
```

```
0.933333333333
```

Task: Why did the accuracy improve compared to the holdout split? What's a disadvantage of the cross validation approach?

Answer:

with CV you use more of the data for training. a disadvantage is the higher computation time, because the model is fitted several times.

1.4 Hyperparameter Optimization

KNN is well known for being easy to use and interpret, howerever to achieve top performance it is not the best choice. In the first Session you got to know the Support Vector Machine (SVM) classifier. In contrast to KNN it has quite a few hyperparameters that need to optimized to achieve top performance. In the next few steps we will optimize SVM hyperparameters by a grid search.

SVMs are not scale invariant, i.e. all features need to be scaled to the same range, for example [-1, 1] or [0, 1]. The most common approach to this is the subtraction of the mean and division by the standard deviation of each feature.

It is one of the most common errors in machine learning to carry out scaling on the whole dataset rather than on training- and test set independently. If this is not considered, information about the distribution of the test set (mean and standard deviation) leaks into the training set and the cross validation is flawed.

A very handy way to overcome this is using the `sklearn.pipeline.Pipeline` class for combining consecutive processing pipelines such as scaling and classification.

Task: Combine an SVM classifier (default parameters) and a `StandardScaler` in a `Pipeline`.

In [12]:
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
'''
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, random_stat

clf = SVC(kernel='linear')

iris_sum=Pipeline([('Standard Scaler', StandardScaler()), ('svc', SVC())])

iris_sum.set_params(svc__C=.1).fit(X_train, y_train)

prediction = iris_sum.predict(X_test)

print(prediction)

print(accuracy(y_test, prediction))
'''
```

```
scaler = StandardScaler()
clf = SVC(kernel='linear')
pipe = Pipeline([("scaler",scaler),('svc',clf)])
pipe.fit(X_train, y_train)
```

```
Out[12]: Pipeline(memory=None,
             steps=[('scaler', StandardScaler(copy=True, with_mean=True, with_std=True)), ('sv
           decision_function_shape='ovr', degree=3, gamma='auto', kernel='linear',
           max_iter=-1, probability=False, random_state=None, shrinking=True,
           tol=0.001, verbose=False))])
```

If you now call `pipe.fit(X_train, y_train)`, the pipeline calculcates mean and standard deviation based on the training data, applies the scaling and trains the SVM.

It's time to tune the SVM's hyperparameters, namely the `C` parameter, the type of kernel and the `gamma` parameter. The naive approach to hyperparameter tuning lies in exhaustively testing all possible combinations of parameters, which is called grid search.

Task: Implement a parameter grid for our three parameters to tune and perform the grid search in a 10-fold cross validation!

Hint: Keep in mind that exhaustive grid search results in high computation times. Use parallelization with the `n_jobs` argument and keep track of time with the `%time` magic command.

```
In [13]: from sklearn.model_selection import GridSearchCV

         parameters = {'svc__kernel':["linear", "poly", "rbf", "sigmoid"], 'svc__C':np.arange(
         #'svc__C':np.arange(
         print(X.shape[0],X.shape[1])
         svc = SVC(kernel='linear')
         scaler = StandardScaler()

         pipe = Pipeline([("scaler",scaler),('svc',clf)])
         gcv = GridSearchCV(pipe, parameters, cv=10)
         print(X.shape,y.shape)
         gcv.fit(X, y)
```

```
150 4
(150, 4) (150,)
```

```
Out[13]: GridSearchCV(cv=10, error_score='raise',
                estimator=Pipeline(memory=None,
             steps=[('scaler', StandardScaler(copy=True, with_mean=True, with_std=True)), ('sv
           decision_function_shape='ovr', degree=3, gamma='auto', kernel='linear',
           max_iter=-1, probability=False, random_state=None, shrinking=True,
           tol=0.001, verbose=False))]),
                fit_params=None, iid=True, n_jobs=1,
                param_grid={'svc__kernel': ['linear', 'poly', 'rbf', 'sigmoid'], 'svc__C': arra
                pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
                scoring=None, verbose=0)
```

Take a closer look at the grid search results. What mean test score can you achieve?

```
In [14]: from evaluation import show_grid_search_results
         np.array(show_grid_search_results(gcv)[:]["mean_train_score"]).max()
         show_grid_search_results(gcv)
```

```
/usr/local/anaconda3/lib/python3.6/site-packages/sklearn/utils/deprecation.py:122: FutureWarni
  warnings.warn(*warn_args, **warn_kwargs)
/usr/local/anaconda3/lib/python3.6/site-packages/sklearn/utils/deprecation.py:122: FutureWarni
  warnings.warn(*warn_args, **warn_kwargs)
/usr/local/anaconda3/lib/python3.6/site-packages/sklearn/utils/deprecation.py:122: FutureWarni
  warnings.warn(*warn_args, **warn_kwargs)
/usr/local/anaconda3/lib/python3.6/site-packages/sklearn/utils/deprecation.py:122: FutureWarni
  warnings.warn(*warn_args, **warn_kwargs)
/usr/local/anaconda3/lib/python3.6/site-packages/sklearn/utils/deprecation.py:122: FutureWarni
  warnings.warn(*warn_args, **warn_kwargs)
/usr/local/anaconda3/lib/python3.6/site-packages/sklearn/utils/deprecation.py:122: FutureWarni
  warnings.warn(*warn_args, **warn_kwargs)
/usr/local/anaconda3/lib/python3.6/site-packages/sklearn/utils/deprecation.py:122: FutureWarni
  warnings.warn(*warn_args, **warn_kwargs)
/usr/local/anaconda3/lib/python3.6/site-packages/sklearn/utils/deprecation.py:122: FutureWarni
  warnings.warn(*warn_args, **warn_kwargs)
/usr/local/anaconda3/lib/python3.6/site-packages/sklearn/utils/deprecation.py:122: FutureWarni
  warnings.warn(*warn_args, **warn_kwargs)
/usr/local/anaconda3/lib/python3.6/site-packages/sklearn/utils/deprecation.py:122: FutureWarni
  warnings.warn(*warn_args, **warn_kwargs)
/usr/local/anaconda3/lib/python3.6/site-packages/sklearn/utils/deprecation.py:122: FutureWarni
  warnings.warn(*warn_args, **warn_kwargs)
/usr/local/anaconda3/lib/python3.6/site-packages/sklearn/utils/deprecation.py:122: FutureWarni
  warnings.warn(*warn_args, **warn_kwargs)
```

Out[14]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_svc__C	\
9	0.000630	0.000016	0.000231	1.206549e-06	3	
16	0.000639	0.000011	0.000229	1.575909e-06	5	
33	0.000635	0.000016	0.000231	1.545313e-06	9	
8	0.000646	0.000029	0.000229	1.234493e-06	3	
28	0.000625	0.000022	0.000228	1.411309e-06	8	
17	0.000632	0.000009	0.000231	1.978874e-06	5	
24	0.000620	0.000022	0.000228	2.126608e-06	7	
25	0.000630	0.000013	0.000232	2.785419e-06	7	
12	0.000637	0.000013	0.000229	3.726430e-06	4	
0	0.000698	0.000199	0.000261	6.762563e-05	1	
20	0.000620	0.000014	0.000229	3.653021e-06	6	
29	0.000630	0.000016	0.000230	1.312169e-06	8	
2	0.000801	0.000044	0.000255	1.986500e-05	1	
21	0.000633	0.000022	0.000232	2.574037e-06	6	
26	0.000762	0.000020	0.000242	3.850958e-06	7	

```
30       0.000757      0.000020         0.000240    1.424339e-06            8
32       0.000607      0.000029         0.000227    2.399515e-06            9
22       0.000759      0.000016         0.000240    1.231727e-06            6
34       0.000756      0.000015         0.000241    2.947243e-06            9
18       0.000750      0.000019         0.000243    5.104831e-06            5
13       0.000632      0.000011         0.000230    2.952927e-06            4
10       0.000749      0.000010         0.000242    3.301904e-06            3
6        0.000760      0.000043         0.000242    1.263618e-06            2
4        0.000623      0.000017         0.000229    2.823927e-06            2
1        0.000649      0.000015         0.000233    2.646446e-06            1
14       0.000744      0.000018         0.000242    3.027540e-06            4
5        0.000640      0.000011         0.000231    9.830250e-07            2
3        0.000734      0.000008         0.000253    3.692333e-06            1
7        0.000724      0.000028         0.000247    1.062234e-06            2
15       0.000682      0.000008         0.000242    7.406342e-07            4
11       0.000702      0.000009         0.000243    7.721965e-07            3
31       0.000679      0.000017         0.000242    3.179918e-06            8
35       0.000680      0.000016         0.000243    3.718795e-06            9
19       0.000681      0.000007         0.000248    1.935595e-05            5
23       0.000679      0.000010         0.000243    3.578583e-06            6
27       0.000677      0.000012         0.000242    3.556275e-06            7

   param_svc__kernel                                  params  \
9               poly     {'svc__C': 3, 'svc__kernel': 'poly'}
16            linear   {'svc__C': 5, 'svc__kernel': 'linear'}
33              poly     {'svc__C': 9, 'svc__kernel': 'poly'}
8             linear   {'svc__C': 3, 'svc__kernel': 'linear'}
28            linear   {'svc__C': 8, 'svc__kernel': 'linear'}
17              poly     {'svc__C': 5, 'svc__kernel': 'poly'}
24            linear   {'svc__C': 7, 'svc__kernel': 'linear'}
25              poly     {'svc__C': 7, 'svc__kernel': 'poly'}
12            linear   {'svc__C': 4, 'svc__kernel': 'linear'}
0             linear   {'svc__C': 1, 'svc__kernel': 'linear'}
20            linear   {'svc__C': 6, 'svc__kernel': 'linear'}
29              poly     {'svc__C': 8, 'svc__kernel': 'poly'}
2                rbf      {'svc__C': 1, 'svc__kernel': 'rbf'}
21              poly     {'svc__C': 6, 'svc__kernel': 'poly'}
26               rbf      {'svc__C': 7, 'svc__kernel': 'rbf'}
30               rbf      {'svc__C': 8, 'svc__kernel': 'rbf'}
32            linear   {'svc__C': 9, 'svc__kernel': 'linear'}
22               rbf      {'svc__C': 6, 'svc__kernel': 'rbf'}
34               rbf      {'svc__C': 9, 'svc__kernel': 'rbf'}
18               rbf      {'svc__C': 5, 'svc__kernel': 'rbf'}
13              poly     {'svc__C': 4, 'svc__kernel': 'poly'}
10               rbf      {'svc__C': 3, 'svc__kernel': 'rbf'}
6                rbf      {'svc__C': 2, 'svc__kernel': 'rbf'}
4             linear   {'svc__C': 2, 'svc__kernel': 'linear'}
1               poly     {'svc__C': 1, 'svc__kernel': 'poly'}
```

```
14              rbf      {'svc__C': 4, 'svc__kernel': 'rbf'}
5              poly     {'svc__C': 2, 'svc__kernel': 'poly'}
3           sigmoid  {'svc__C': 1, 'svc__kernel': 'sigmoid'}
7           sigmoid  {'svc__C': 2, 'svc__kernel': 'sigmoid'}
15          sigmoid  {'svc__C': 4, 'svc__kernel': 'sigmoid'}
11          sigmoid  {'svc__C': 3, 'svc__kernel': 'sigmoid'}
31          sigmoid  {'svc__C': 8, 'svc__kernel': 'sigmoid'}
35          sigmoid  {'svc__C': 9, 'svc__kernel': 'sigmoid'}
19          sigmoid  {'svc__C': 5, 'svc__kernel': 'sigmoid'}
23          sigmoid  {'svc__C': 6, 'svc__kernel': 'sigmoid'}
27          sigmoid  {'svc__C': 7, 'svc__kernel': 'sigmoid'}

    split0_test_score  split1_test_score  split2_test_score       ...         \
9            1.000000           1.000000                1.0       ...
16           1.000000           1.000000                1.0       ...
33           1.000000           1.000000                1.0       ...
8            1.000000           1.000000                1.0       ...
28           1.000000           1.000000                1.0       ...
17           1.000000           1.000000                1.0       ...
24           1.000000           1.000000                1.0       ...
25           1.000000           1.000000                1.0       ...
12           1.000000           1.000000                1.0       ...
0            1.000000           0.933333                1.0       ...
20           1.000000           1.000000                1.0       ...
29           1.000000           1.000000                1.0       ...
2            1.000000           0.933333                1.0       ...
21           1.000000           1.000000                1.0       ...
26           1.000000           1.000000                1.0       ...
30           1.000000           1.000000                1.0       ...
32           1.000000           1.000000                1.0       ...
22           1.000000           1.000000                1.0       ...
34           1.000000           1.000000                1.0       ...
18           1.000000           1.000000                1.0       ...
13           1.000000           1.000000                1.0       ...
10           1.000000           0.933333                1.0       ...
6            1.000000           0.933333                1.0       ...
4            1.000000           1.000000                1.0       ...
1            1.000000           0.933333                1.0       ...
14           1.000000           0.933333                1.0       ...
5            1.000000           0.933333                1.0       ...
3            0.733333           0.866667                1.0       ...
7            0.733333           0.800000                1.0       ...
15           0.733333           0.800000                1.0       ...
11           0.733333           0.800000                1.0       ...
31           0.733333           0.733333                1.0       ...
35           0.733333           0.733333                1.0       ...
19           0.733333           0.733333                1.0       ...
23           0.733333           0.733333                1.0       ...
```

```
27          0.733333          0.733333               1.0       ...

    split2_train_score  split3_train_score  split4_train_score  \
9             0.970370            0.970370            0.962963
16            0.970370            0.970370            0.985185
33            0.962963            0.970370            0.970370
8             0.962963            0.970370            0.985185
28            0.970370            0.970370            0.985185
17            0.962963            0.962963            0.955556
24            0.970370            0.970370            0.985185
25            0.962963            0.970370            0.962963
12            0.970370            0.970370            0.985185
0             0.970370            0.962963            0.985185
20            0.970370            0.970370            0.985185
29            0.962963            0.970370            0.970370
2             0.970370            0.977778            0.970370
21            0.962963            0.970370            0.955556
26            0.977778            0.985185            0.985185
30            0.985185            0.985185            0.985185
32            0.970370            0.970370            0.985185
22            0.977778            0.985185            0.985185
34            0.985185            0.985185            0.985185
18            0.985185            0.985185            0.985185
13            0.962963            0.962963            0.955556
10            0.970370            0.977778            0.977778
6             0.970370            0.977778            0.977778
4             0.962963            0.970370            0.977778
1             0.925926            0.940741            0.948148
14            0.985185            0.985185            0.985185
5             0.962963            0.962963            0.962963
3             0.881481            0.896296            0.896296
7             0.837037            0.881481            0.888889
15            0.874074            0.851852            0.837037
11            0.851852            0.859259            0.874074
31            0.859259            0.851852            0.859259
35            0.866667            0.851852            0.859259
19            0.866667            0.859259            0.837037
23            0.866667            0.859259            0.844444
27            0.859259            0.851852            0.859259

    split5_train_score  split6_train_score  split7_train_score  \
9             0.977778            0.970370            0.970370
16            0.970370            0.977778            0.970370
33            0.977778            0.985185            0.970370
8             0.977778            0.970370            0.962963
28            0.970370            0.985185            0.970370
17            0.977778            0.977778            0.970370
24            0.970370            0.985185            0.970370
```

25	0.970370	0.985185	0.970370
12	0.977778	0.977778	0.970370
0	0.977778	0.977778	0.970370
20	0.970370	0.985185	0.970370
29	0.977778	0.985185	0.970370
2	0.977778	0.985185	0.977778
21	0.970370	0.977778	0.970370
26	0.992593	0.985185	0.985185
30	0.992593	0.985185	0.985185
32	0.970370	0.985185	0.970370
22	0.992593	0.985185	0.985185
34	0.992593	0.985185	0.985185
18	0.992593	0.985185	0.985185
13	0.977778	0.970370	0.970370
10	0.977778	0.985185	0.977778
6	0.985185	0.985185	0.970370
4	0.970370	0.970370	0.962963
1	0.955556	0.955556	0.948148
14	0.985185	0.977778	0.985185
5	0.977778	0.962963	0.970370
3	0.881481	0.903704	0.911111
7	0.866667	0.866667	0.874074
15	0.844444	0.874074	0.837037
11	0.851852	0.866667	0.866667
31	0.844444	0.874074	0.866667
35	0.844444	0.874074	0.859259
19	0.844444	0.874074	0.837037
23	0.844444	0.874074	0.859259
27	0.844444	0.874074	0.866667

	split8_train_score	split9_train_score	mean_train_score	std_train_score
9	0.962963	0.970370	0.968889	0.004444
16	0.970370	0.970370	0.972593	0.004743
33	0.970370	0.970370	0.971111	0.006153
8	0.962963	0.962963	0.968148	0.007444
28	0.970370	0.970370	0.974074	0.005972
17	0.962963	0.977778	0.967407	0.007554
24	0.970370	0.970370	0.973333	0.005926
25	0.970370	0.970370	0.971111	0.006153
12	0.970370	0.970370	0.971852	0.006458
0	0.962963	0.970370	0.971111	0.006988
20	0.970370	0.970370	0.973333	0.005926
29	0.970370	0.970370	0.971111	0.006153
2	0.970370	0.977778	0.974815	0.004914
21	0.962963	0.970370	0.967407	0.005926
26	0.985185	0.985185	0.984444	0.003989
30	0.985185	0.985185	0.985185	0.003313
32	0.970370	0.970370	0.973333	0.005926

```
22          0.985185          0.985185          0.984444          0.003989
34          0.985185          0.985185          0.985185          0.003313
18          0.985185          0.985185          0.985926          0.002222
13          0.962963          0.970370          0.966667          0.005972
10          0.970370          0.970370          0.975556          0.004743
6           0.970370          0.970370          0.975556          0.005785
4           0.962963          0.962963          0.966667          0.004969
1           0.948148          0.918519          0.944444          0.012058
14          0.985185          0.977778          0.982222          0.003629
5           0.962963          0.962963          0.964444          0.005543
3           0.866667          0.903704          0.896296          0.014815
7           0.859259          0.859259          0.868148          0.015108
15          0.859259          0.866667          0.857037          0.014074
11          0.837037          0.851852          0.859259          0.012395
31          0.859259          0.866667          0.860000          0.008413
35          0.859259          0.866667          0.860000          0.008413
19          0.851852          0.874074          0.857778          0.015108
23          0.859259          0.874074          0.860741          0.010887
27          0.866667          0.866667          0.860741          0.008638

[36 rows x 32 columns]
```

As the number of hyperparameter is usually much higher than here, a naive grid search is often not feasible in practice. A good alternative lies in a randomized search, in which the parameter space is sampled coarsely.

Task: Implement the same workflow as above withe the same parameter range but using `RandomizedSarchCV`. Can you achieve comparable performance in shorter time?

```
In [15]: from sklearn.model_selection import RandomizedSarchCV

         parameters = {'svc__kernel':["linear", "poly", "rbf", "sigmoid"],'svc__C':np.arange(1
         #'svc__C':np.arange(
         '''
         print(X.shape[0],X.shape[1])
         svc = SVC(kernel='linear')
         scaler = StandardScaler()

         pipe = Pipeline([("scaler",scaler),('svc',clf)])
         gcv = RandomizedSarchCV(pipe, parameters, cv=10)
         print(X.shape,y.shape)
         gcv.fit(X, y)
         '''

         print(X.shape[0],X.shape[1])
         svc = SVC(kernel='linear')
         scaler = StandardScaler()
         pipe = Pipeline([("scaler",scaler),('svc',clf)])
         gcv = RandomizedSearchCV(pipe, parameters, cv=10)
```

```
print(X.shape,y.shape)
gcv.fit(X, y)
np.array(show_grid_search_results(gcv)[:]["mean_train_score"]).max()
show_grid_search_results(gcv).shape
```

```
---------------------------------------------------------------------------

ImportError                               Traceback (most recent call last)

<ipython-input-15-e4243fd0cbc9> in <module>()
----> 1 from sklearn.model_selection import RandomizedSarchCV
      2
      3 parameters = {'svc__kernel':["linear", "poly", "rbf", "sigmoid"],'svc__C':np.arang
      4 #'svc__C':np.arange(
      5 '''


ImportError: cannot import name 'RandomizedSarchCV'
```

You probably found out that it is possible to achieve literally the same mean test score using a randomizes search in just the fraction of the time needed for al full grid search.

Task: Is the mean test score a good estimation of the performance we can expect on unseen data?

Answer:

1.5 Evaluation Measures

To make this more interesting, the iris dataset was slightly modified to better resemble real-world datasets. The following modified Iris dataset is smaller, there are only two classes present and the prevalence for the two classes is imbalanced.

```
In [ ]: data = load_iris(imbalanced_binary=True)
        _ = sns.pairplot(data, hue='species', vars=data.columns[:-1])
```

Task: Solve the binary classification problem by a KNN classifier, optimize the number of neighbors by 5 fold cross validation and evaluate accuracy on the test set using the best `n_neighbors`.

```
In [ ]: X_train, X_test, y_train, y_test = train_test_split(data[['sepal_length', 'sepal_width
```

Not too bad. You probably achieved an accuracy near 90%. After this first result we should take a more detailed look at the performance using a confusion matrix, i.e. find out the number of true positives, true, negatives, false positives and false negatives.

```
In [ ]:
```

Apparently, we only had four positive samples in the test set, out of which one half was classified incorrectly.

Task: What do you conclude about accuracy as a metric? Whats the accuracy of just labelling all samplese as negative?

In [ ]:

** Solution:**

1.5.1 ROC Curves

A better way to assess classification performance for imbalanced problems lies in looking at the *true positive rate* and *false positive rate* (also known as sensitivity and specifity).

These two can even be plotted against each other, which results in a receiver operating curve (ROC). The higher the area under the ROC curve (AUC) the better the classifier.

This is especially useful in medical image processing, a context in which we usually have data from a population out of which a small fraction has a particular disease that need to be detected.

Task: Whats are the *true positive rate*, *false positive rate* and *AUC* of our classifier? Plot the ROC curve!

```
In [ ]: from sklearn.metrics import roc_auc_score, roc_curve
```

1.6 Yayy, high accuracy!... Just by chance?

Task: What if we were to consider only two features for classification, which ones would you choose? Would the accuracy be 'significantly' different depending on the choice of features?

Answer:

We will use the scipy.stats module for simple statistical tests.

```
In [ ]: from scipy import stats
```

1.7 Student's t-test

Let us consider two feature groups and see if we get similar accuracies or not: 1. only sepal length and petal length 2. only sepal width and petal width

```
In [16]: data = load_iris()
         X = data[['sepal_length', 'sepal_width', 'petal_length', 'petal_width']].as_matrix()
         X_length = data[['petal_length', 'sepal_length']].as_matrix() # Considering only leng
         X_width = data[['sepal_width', 'petal_width']].as_matrix() # Considering only widths

         y = data['species'].values

         scores = np.zeros(100)
         scores_length = np.zeros(100)
         scores_width = np.zeros(100)

         # Calculate accuracy when considering all the features
         for i in range(100):
```

```
        X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=i, test_siz

        model = KNN(n_neighbors=2)
        model.fit(X_train, y_train)

        y_pred = model.predict(X_test)
        scores[i] = accuracy_score(y_test, y_pred)
    print('Mean accuracy considering all the features = {:2.3}'.format(scores.mean()))

    # Calculate accuracy when considering only lengths
    for i in range(100):
        X_length_train, X_length_test, y_length_train, y_length_test = train_test_split(X_

        model = KNN(n_neighbors=2)
        model.fit(X_length_train, y_length_train)

        y_length_pred = model.predict(X_length_test)
        scores_length[i] = accuracy_score(y_length_test, y_length_pred)

    print('Mean accuracy considering petal_length and sepal_length only = {:2.3}'.format(s

    # Calculate accuracy when considering only widths
    for i in range(100):
        X_width_train, X_width_test, y_width_train, y_width_test = train_test_split(X_widt

        model = KNN(n_neighbors=2)
        model.fit(X_width_train, y_width_train)

        y_width_pred = model.predict(X_width_test)
        scores_width[i] = accuracy_score(y_width_test, y_width_pred)
    print('Mean accuracy considering petal_width and sepal_width only = {:2.3}'.format(sco

/usr/local/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:2: FutureWarning: Metho

/usr/local/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:3: FutureWarning: Metho
  This is separate from the ipykernel package so we can avoid doing imports until
/usr/local/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:4: FutureWarning: Metho
  after removing the cwd from sys.path.


---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-16-5da890fb7f3c> in <module>()
 18
 19     y_pred = model.predict(X_test)
```

```
---> 20     scores[i] = accuracy_score(y_test, y_pred)
     21 print('Mean accuracy considering all the features = {:2.3}'.format(scores.mean()))
     22


NameError: name 'accuracy_score' is not defined
```

1.7.1 2-sample t-test: testing for difference across populations

The means of both the groups are different. To test if this is significant, we perform a 2-sample t-test with scipy.ttest_ind()

```
In [17]: stats.ttest_ind(scores_length, scores_width)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-17-b4b0b6e81aa6> in <module>()
----> 1 stats.ttest_ind(scores_length, scores_width)


NameError: name 'stats' is not defined
```

If the observed p-value is smaller than the threshold value (typically 0.01, 0.05, 0.1), we reject the null hypothesis of equal means. In this case, however, we fail to find statistical difference.

Task: Are the groups X_petal and X_sepal significantly different from X?

```
In [ ]:
```

Conclusion

```
X_petal is significantly different from X
X_sepal is significantly different from X
```

1.7.2 Paired t-test: repeated measurements

Note that the measurements for petal and sepal are made on the same flowers! In such a case where each subject is measured twice, we can use a paired sample t-test.

```
In [18]: stats.ttest_rel(scores_length, scores_width)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)
```

```
    <ipython-input-18-34e7c89dd2a5> in <module>()
----> 1 stats.ttest_rel(scores_length, scores_width)


NameError: name 'stats' is not defined
```

Another thing we overlooked is the fact that t-tests assume a gaussian distribution of the data. But is this true for our data? Let's investigate:

In [19]:
```
# Plot the histograms to see the data distribution

bins = 25
data_x = (scores, scores_length, scores_width)
x_labels = ('scores', 'scores_length', 'scores_width')
xlims = [[0.85, 1]]*3
ylims = [[0, 30]]*3

for j, (x, xlim, ylim, x_label) in enumerate(zip(data_x, xlims, ylims, x_labels)):
    ax = plt.subplot(1, 3, j+1)
    ax.hist(np.reshape(x,(-1,1)), bins = bins)
    ax.set_xlim(xlim)
    ax.set_ylim(ylim)
    ax.set_xlabel(x_label)

ax = plt.subplot(1,3,1)
ax.set_ylabel('frequency');
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-19-879b3056a54d> in <module>()
  8
  9 for j, (x, xlim, ylim, x_label) in enumerate(zip(data_x, xlims, ylims, x_labels)):
---> 10     ax = plt.subplot(1, 3, j+1)
 11     ax.hist(np.reshape(x,(-1,1)), bins = bins)
 12     ax.set_xlim(xlim)


NameError: name 'plt' is not defined
```

Oops! Our data does not have a gaussian distibution! To take this into account, we must use Wilcoxon signed-rank test, with the help of scipy.stats.wilcoxon().

Task: Calculate the wilcoxon signed-rank test for the groups above. Draw a conclusion!

In [ ]:
```
```

pvalue > 0.1 Hence we accept the Null Hypothesis of equal means, which concludes that we fail to find significant differences at p=0.1

Note: The corresponding test for non-paired case is the Mann-Whitney U test (scipy.stats.mannwhitneyu)

1.7.3 Feedback

That's it, we're done

If you have any suggestions on how we could improve this session, please let us know in the following cell. What did you particularly like or dislike? Did you miss any contents?