Overview

This tutorial will focus on Logistic Regression

We will include some additional topics, including:

- Handling categorical attributes
- Logistic regression
- Training for maximum likelihood
- Evaluation of classification models

The lecture relies on the the following proficiencies and tools:

- Python programming
- Vector computations via the numpy module
- Data handling using the pandas module
- Plotting using matplotlib
- Training and using Machine Learning model via scikit-learn

You will need them only if you plan to handle these tasks yourself

Categorical Attributes

Categorical Attributes

Let's switch to a different dataset (a toy one)

- We want to train a model to choose whether to go out and play
- ...Based on weather conditions

Loading the Data

The dataset is in the `weather.csv` file from the `data` folder

```
In [2]: !ls data

lr_test.txt  lr_train.txt  real_estate.csv  weather.csv
```

```
In [3]: import pandas as pd
        data = pd.read_csv('data/weather.csv', sep=',')
        data.head()
```

Out[3]:

	outlook	temperature	humidity	windy	play
0	sunny	85	85	False	no
1	sunny	80	90	True	no
2	overcast	83	86	False	yes
3	rainy	70	96	False	yes
4	rainy	68	80	False	yes

- Several attributes do not have a numeric value
- Instead, their value is discrete with no clear ordering, i.e. categorical

We need a numeric encoding to handle this data with linear models

Encoding Binary Attributes

Binary attributes can be encoded with the values 0 and 1

This is the case for the columns "windy" and "play"

- First, we tell pandas that the columns have a categorical type

```
In [4]: windy = data['windy'].astype('category')
        play = data['play'].astype('category')
        windy.head()

Out[4]: 0    False
        1     True
        2    False
        3    False
        4    False
        Name: windy, dtype: category
        Categories (2, object): [False, True]
```

- Categorical data is still displayed as a string
- ...But internally it is encoded as an integer

Encoding Binary Attributes

Next, we replace the values with their integer code

We will store the results in a copy of the original table

In [5]:
```
data2 = data.copy() # We prepare a cop for the numeric encodings
data2['windy'] = windy.cat.codes
data2['play'] = play.cat.codes
data2.head()
```

Out[5]:

	outlook	temperature	humidity	windy	play
0	sunny	85	85	0	0
1	sunny	80	90	1	0
2	overcast	83	86	0	1
3	rainy	70	96	0	1
4	rainy	68	80	0	1

- Now it is apparent that "windy" and "play" have become numbers

Encoding Discrete Attributes

We could use the same approach for discrete attribute in general

E.g. for the attribute "outlook" in our table

- That would yield a numeric integer encoding
- ...Which implies an ordering among the values (e.g. rainy < overcast < sunny)
- When no such ranking exists, this is a bad idea

In these cases, it is better to adopt a one-hot encoding

- We introduce a column for each value v_k of the attribute xj
- The column contains a 1 iff $x_j = v_k$, and 0 otherwise

For example, "sunny | sunny | overcast" becomes:

rainy	overcast	sunny
0	0	1
0	0	1
0	1	0

Encoding Discrete Attributes

We can obtain a one-hot encoding in pandas via the `get_dummies` method

In [6]:
```
data2 = pd.get_dummies(data2)
data2.head()
```

Out[6]:

	temperature	humidity	windy	play	outlook_overcast	outlook_rainy	outlook_sunny
0	85	85	0	0	0	0	1
1	80	90	1	0	0	0	1
2	83	86	0	1	1	0	0
3	70	96	0	1	0	1	0
4	68	80	0	1	0	1	0

- The method by default processes all columns with categorical or object type
- strings in csv files are often parsed as "object" columns
- In truth, `get_dummies` can also handle the special case of binary variables
- ...But I wanted to show you how to obtain an integer encoding, too :-)

Logistic Regression

Logistic Regression

Our goal is to predict the value of "play", i.e. a categorical attribute

We say that we are dealing with a classification problem and:

- We compute the output of the linear model as usual:

$$g(x; w) = \sum_{j=1} w_j x_j + w_0$$

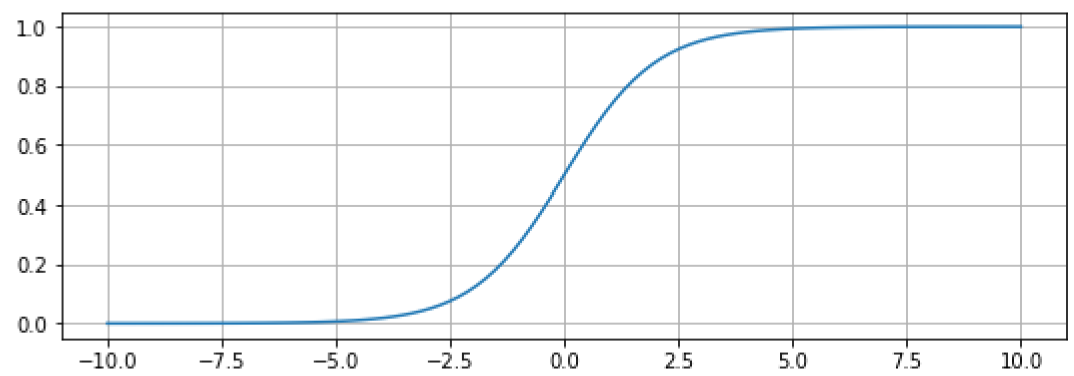
- ...But then we feed it to a logistic function:

$$f(x; w) = \frac{1}{1 + e^{-g(x;w)}}$$

Logistic Regression

The logistic function is a type of sigmoid function

In [7]:
```
import numpy as np
from matplotlib import pyplot as plt
x = np.linspace(-10, 10, 100)
plt.figure(figsize=(9, 3))
plt.plot(x, 1 / (1 + np.exp(-x)))
plt.grid(':')
```



- Due to its use, this approach is known as logistic regression

Logistic Regression

Why using the logistic function?

- We can view the model output as a probability distribution
- Specifically, as the probability of the class being "1"

With this convention, the target can also be interpreted as a probability

```
In [8]: data2['play'].head()

Out[8]: 0    0
        1    0
        2    1
        3    1
        4    1
        Name: play, dtype: int8
```

We view:

- $y_i = 0$ as "the probability of the class being 1 is equal to 0"
- $y_i = 1$ as "the probability of the class being 1 is equal to 1"

Likelihood Function

We now pretend that our model $f(x;w)$ is the source of our data

E.g. if $f(x;w) = 0.7$

- We will generate a 1 with 70% chance
- We will generate a 0 with 30% chance

Why is this interesting?

Now we can measure the chance that the model makes the right guess:

- When we have $y_i = 1$, i.e. the label is 1
  - We will guess right with a $f(x;w)$ probability, wrong with $1 - f(x;w)$
- When we have $y_i = 0$, i.e. the label is 0
  - We will guess right with a $1 - f(x;w)$ probability, wrong with $f(x;w)$

Likelihood Function

In summary

- When we have $y_i = 1$, we want $f(x; w)$ to be high
- When we have $y_i = 0$, we want $f(x; w)$ to be low

If we repeat for all examples (assuming statistical independence)...

We get the the probability of generating our data.

Likelihood Function

We both classes, we take yet another product

$$L(w) = \prod_{y_i=1} f(x_i; w) \prod_{y_i=0} (1 - f(x_i; w))$$

- The is sort of a probability, but is associated to our model, not to the data itself
- ...And it also depends on the parameters w

To make the distinction clearer, we call it a likelihood function

We want to train a model that is a likely source for our data

This means that we can choose the weights by solving:

$$\text{argmax}_w \log L(w)$$

- I.e. to maximize the likelihood of the data
- This often done via Gradient Descent

Maximum Likelihood Estimation

MLE is very important in many Machine Learning approaches

- It provides a mathematical foundation for the training process
- It applies to linear regression, too!
- ...Since the MSE can be interpreted in terms of likelihood

It even works when targets/labels are absent

- I.e. for unsupervised learning scenarios

Using Logistic Regression

Using Logistic Regression in scikit-learn is actually easy

We begin by splitting input/output data as usual:

In [9]:
```
cols_in = [c for c in data2.columns if c != 'play']

X = data2[cols_in]
y = data2['play']
y.head() # We have a table here, but a vector would also work
```

Out[9]:
```
0    0
1    0
2    1
3    1
4    1
Name: play, dtype: int8
```

Then the training and test set:

In [10]:
```
from sklearn.model_selection import train_test_split
X_tr, X_ts, y_tr, y_ts = train_test_split(X, y, test_size=0.34, random_state=0)
```

Using Logistic Regression

Then, we build a `LogisticRegression` model

In [11]:
```
from sklearn.linear_model import LogisticRegression

m = LogisticRegression()
```

...And we call the `fit` method as usual:

In [12]:
```
m.fit(X_tr, y_tr);
```

Finally, we can obtain out predictions:

In [13]:
```
y_pred_tr = m.predict(X_tr)
y_pred_ts = m.predict(X_ts)
```

A Better Look at the Predictions

By default, the prediction is the class with the largest probability

In [14]:
```
y_pred_tr
```

Out[14]:
```
array([0, 1, 0, 0, 1, 0, 0, 1, 1], dtype=int8)
```

- If we are interested in the raw probability values...
- ...We can call the `predict_proba` method:

In [15]:
```
y_prob_tr = m.predict_proba(X_tr)
y_prob_tr[:5]
```

Out[15]:
```
array([[0.726788  , 0.273212  ],
       [0.3675285 , 0.6324715 ],
       [0.77878844, 0.22121156],
       [0.77593317, 0.22406683],
       [0.48442042, 0.51557958]])
```

- Scikit-learn gives us the predicted probability of both classes
- Hence, we get two separate columns

Evaluation

We can evaluate the results using metrics

There are four basic metrics for binary classification:

- Number of True Positives, i.e. $TP = \sum_{y_i=1} \tilde{f}(x_i; w)$
- Number of True Negatives, i.e. $TN = \sum_{y_i=0}(1 - \tilde{f}(x_i; w))$
- Number of False Positives, i.e. $FP = \sum_{y_i=0} \tilde{f}(x_i; w)$
- Number of False Negatives, i.e. $FN = \sum_{y_i=1}(1 - \tilde{f}(x_i; w))$

In all cases $\tilde{f}(x_i; w)$ is the most probable class for the example x_i

Evaluation

From these we can derive a few more complex metrics

The model (binary) accuracy is defined as:

$$ACC = \frac{TP + TN}{m}$$

- I.e. the fraction of examples that is correctly classified
- The accuracy ranges over the interval $[0, 1]$

In [16]:
```
from sklearn.metrics import accuracy_score

print(f'Accuracy on the training set: {accuracy_score(y_tr, y_pred_tr):.3}')
print(f'Accuracy on the test set: {accuracy_score(y_ts, y_pred_ts):.3}')
```

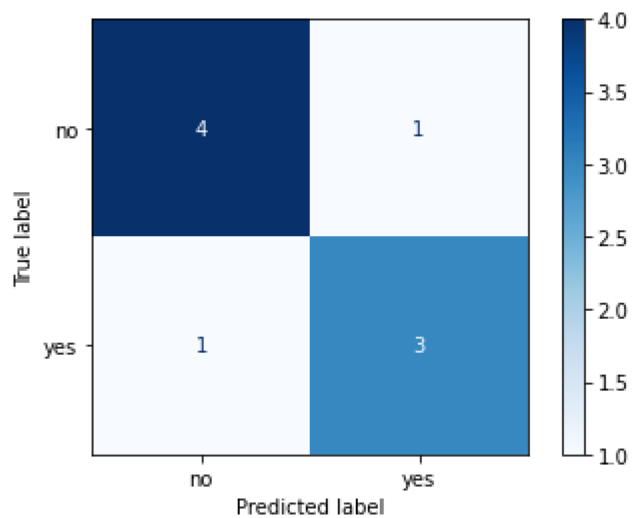
```
Accuracy on the training set: 0.778
Accuracy on the test set: 0.8
```

Evaluation

...Or we can plot all basic metrics via a confusion matrix

Here's the one for the training set:

In [17]:
```python
from sklearn.metrics import plot_confusion_matrix
plot_confusion_matrix(m, X_tr, y_tr, display_labels=play.cat.categories, cmap='Blues');
```

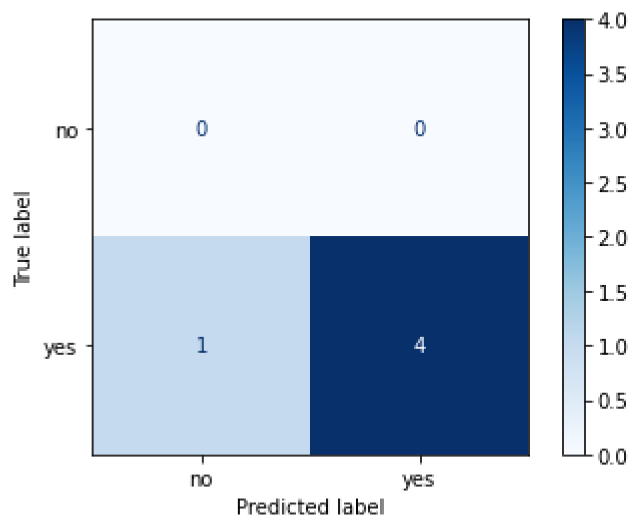


Evaluation

...Or we can plot all basic metrics via a confusion matrix

...And here for the test set:

```
In [18]: from sklearn.metrics import plot_confusion_matrix
         plot_confusion_matrix(m, X_ts, y_ts, display_labels=play.cat.categories, cmap='Blues');
```



Conclusions and Take-Home Messages