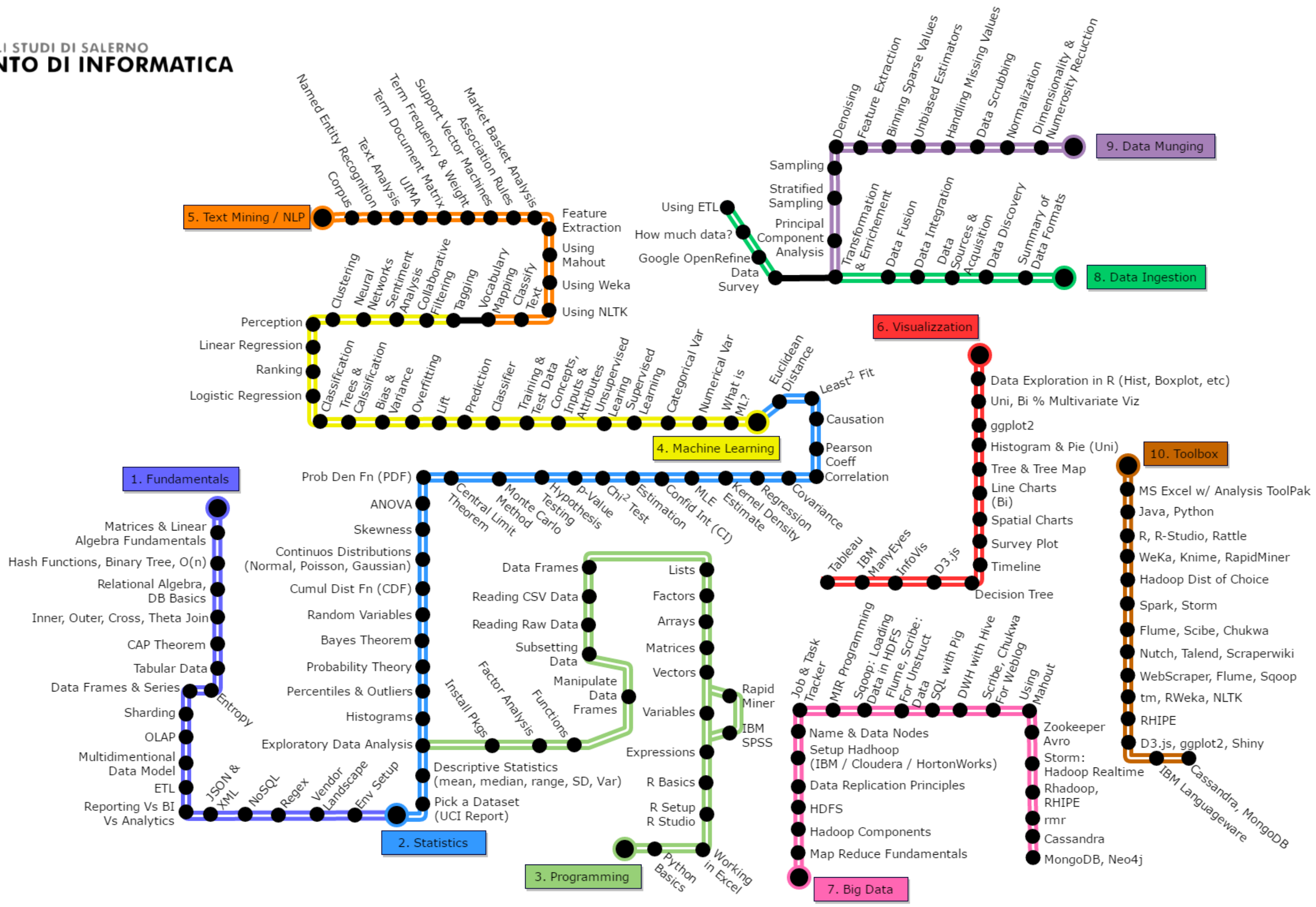
UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA



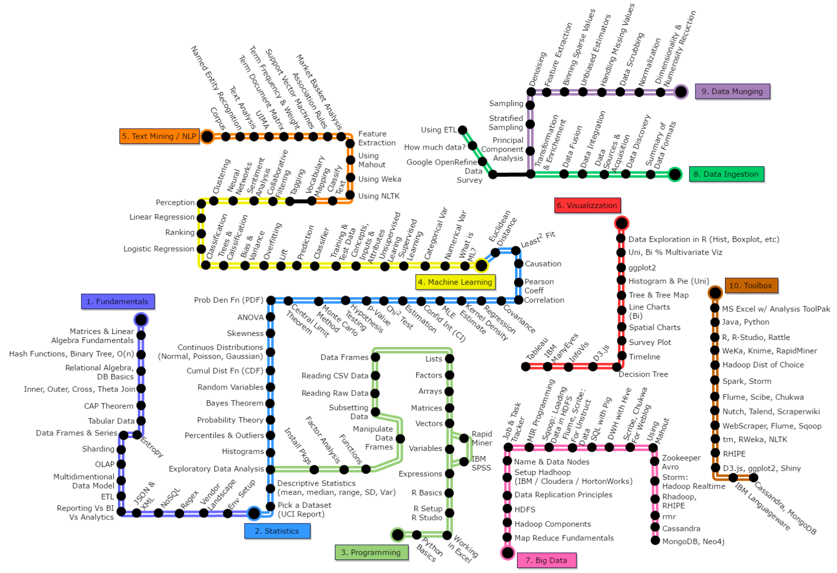
Fondamenti di Data Science e Machine Learning

Classification (Chapter 3 Geron’s book)

Prof. Giuseppe Polese, aa 2024-25

Outline

- **Introduction to Classification/Regression**
  - Training vs test set split
  - Classification/Regression rules
  - Types of classifiers
- Evaluation of Classifiers
  - Evaluation criteria
  - Model evaluation
    - Cross-Validation
    - Confusion matrices
    - Accuracy, Precision, Recall, and F1-score, the ROC Curve
- Classification algorithms: Naïve Bayes
- Other types of classification



Classification

- Problem's data:
  - Set of classes (values of a categorical attribute)
  - Set of objects labelled with the name of the class they belong to (*training set*)
- Problem:
  - Find the descriptive profile of each class, exploiting the features of the training set data, such that it is possible to assign objects contained into a *test set* to a suitable class

How it works

- Training set contained in main memory
- In current DBs there might be Mbyte of training data available
  - Significative dimensions of the training set might potentially improve the accuracy of classification

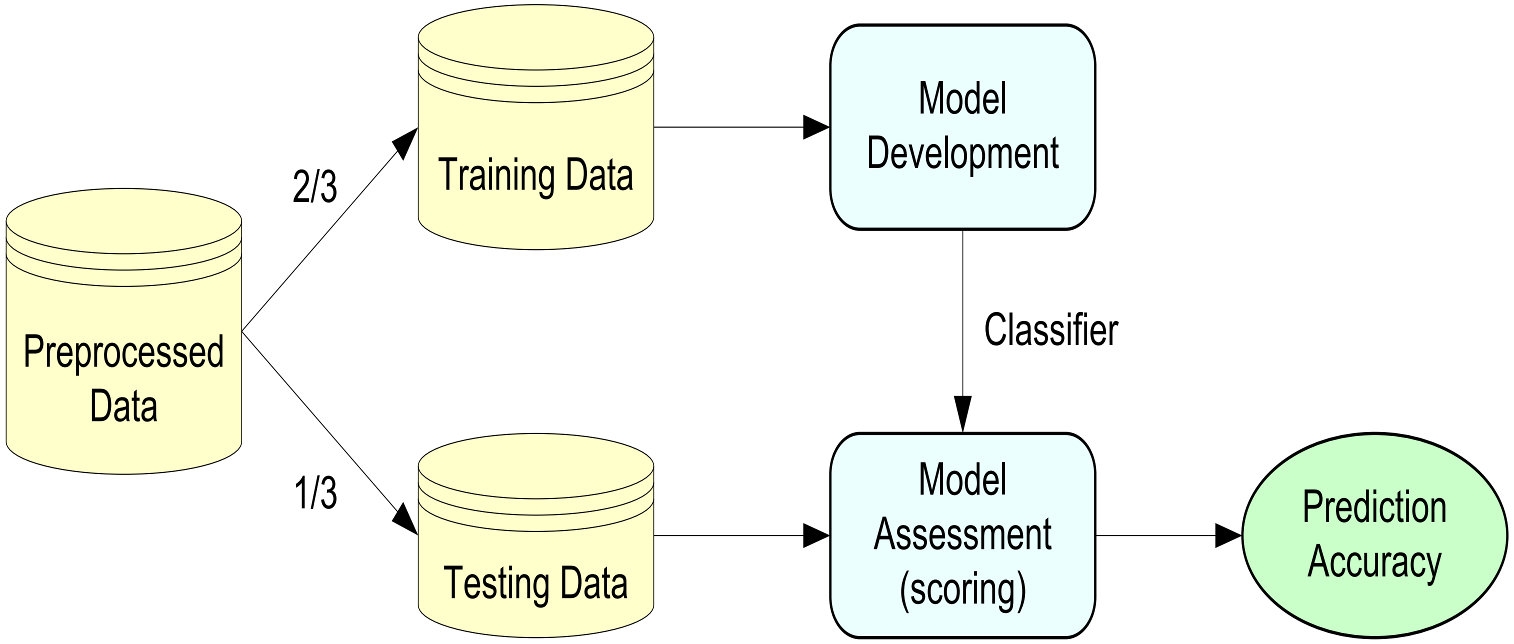
Training set vs Test set

The reason we split our data into training and test sets is that we are interested in measuring how well our model generalizes to new, previously unseen, data

- We are not interested in how well our model fits the training set
- We are interested in how well it can make predictions for data that was not observed during training

Split of Training and Test Set

- Simple Split
  - Partition data into two mutually exclusive sets, ~70%- ~80% training e ~20%- ~30% testing



Classification and regression rules

- Classification rule:

  $P_1(X_1) \wedge \ldots \wedge P_k(X_k) \Longrightarrow Y = c$

  Y *dependent attribute*

  X_i *predictive attributes*

  P_i (X_i) condition on attribute X_i

- Two types of attributes:
  - numeric
  - categoric (enumeration types)

Numeric and Categoric Attributes

- If the dependent attribute is categoric, then we have a *classification rule*
- If it is numeric, then we have a *regressione rule*
- If X_i is numeric, $P_i(X_i)$ generally consists of $l_i \leq X_i \leq g_i$, with l_i e g_i belonging to the domain of X_i
- If X_i is categoric, $P_i\,(X_i)$ consists of $X_i \in \{v_{i,\ldots,}\, v_n\}$

Example

- InfoInsurance(age:int, car_type:string,risk:{high,low})

- Classification rule

$$\text{age} > 25 \wedge \text{car\_type} \in \{\text{Sport, economy}\} \Longrightarrow \text{risk = high}$$

Support and confidence of rules

- As for association rules, also for classification and regression rules it is possible to define measures like support and confidence
- The support of a condition C is the percentage of tuples satisfying C
- The support of a classification/regression rule $C_1 \Rightarrow C_2$ corresponds to the support of $C_1 \wedge C_2$
- The confidence of a rule $C_1 \Rightarrow C_2$ is derived by diving the support of the rule by the support of C_1

Applications

- Classification/regression rules are typically used in applications such as:
  - Classification of scientific results, in which objects must be classified based on experimental data
  - Risk analysis
  - Economic forecasts

Types of classifiers

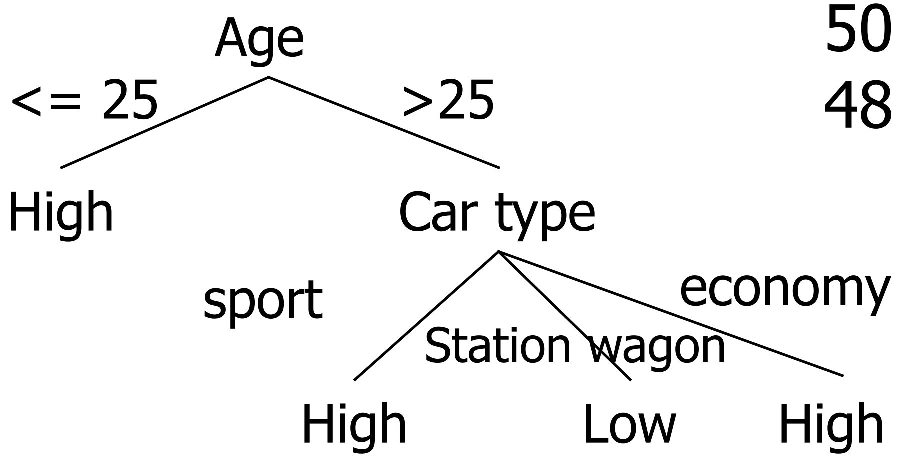
- There exist several types of classifiers:
  - Decision trees classifiers
  - Bayesian classifiers
  - Statistic classifiers
  - Neural net based classifiers
  - ……….

Decision Tree classifiers

- Decision trees represent a widely used approach to classification
- They permit to represent a set of classification rules through a Tree
- Caracteristics:
  - Fast with respect to other methods
    - Easy to interprete through classification rules (one for each path in the tree)
    - Can be easily converted in SQL statements to inquire the database

Example

AGE	CAR TYPE	RISK
40	Station wagon	low
65	Sport	high
20	Economy	high
25	Sport	high
50	Station wagon	low
48	Economy	high



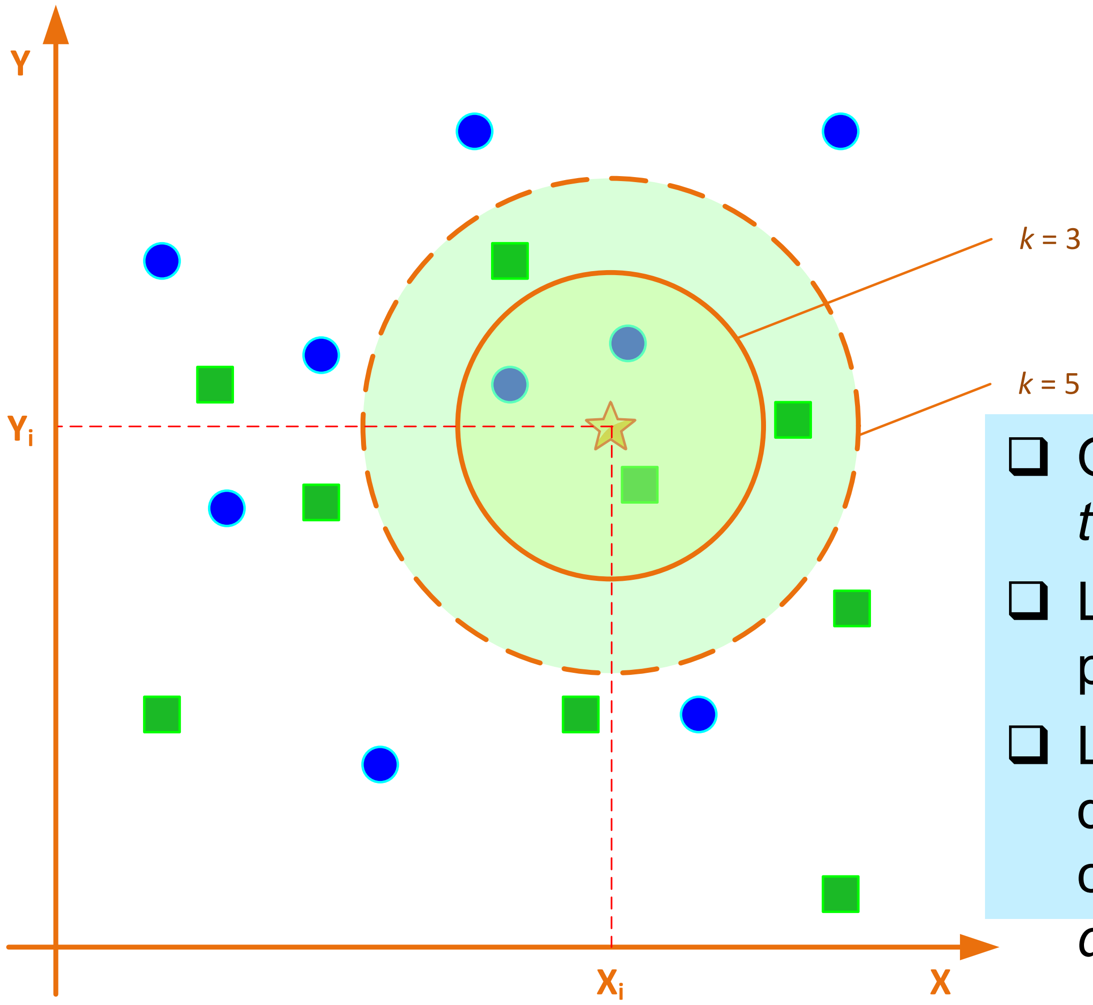
Metodo k-Nearest Neighbor (k-NN)

- E’ uno dei più semplici algoritmi di machine learning
- Un’alternativa alle reti neurali ed all’algoritmo di machine leaning Support Vector Machine (SVM), computazionalmente troppo complessi
- *k*-NN è un metodo predittivo utilizzabile sia per la classificazione che per la regressione
- E’ un approccio di learning di tipo instance-based (o lazy learning) – gran parte del lavoro è svolto a tempo di classificazione anziché di modellazione
- *k* : il numero di vicini usati per la classificazione

Funzionamento *di* k-NN

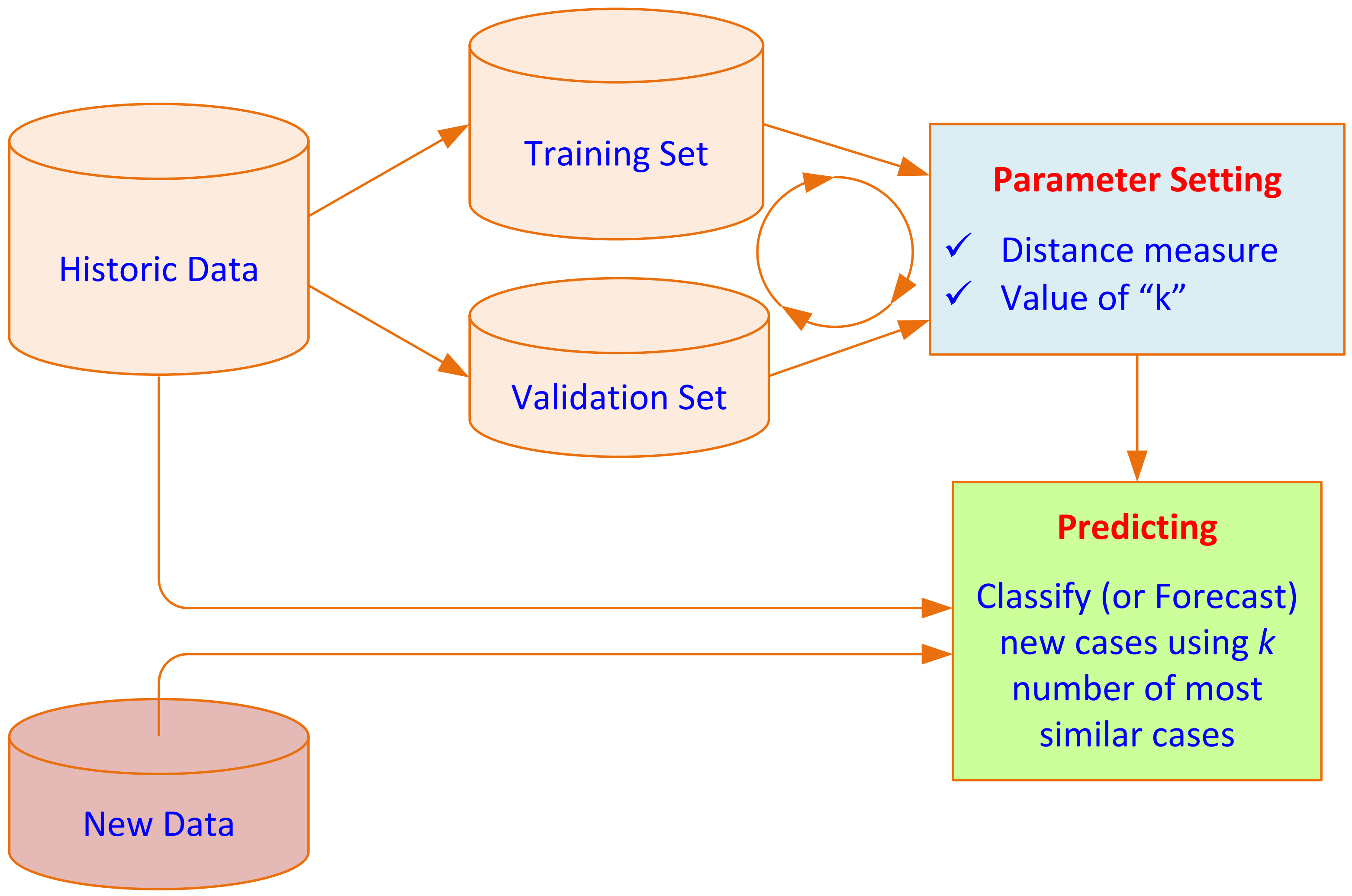
- Ogni nuovo oggetto viene classificato usando la maggioranza dei suoi *k* oggetti più vicini
- L’oggetto viene quindi assegnato alla classe più ricorrente tra i *k* oggetti più vicini
- Se k=1 l’oggetto viene semplicemente inserito nella classe dell’oggetto più vicino
- Oltre al parametro *k* occorre selezionare una funzione di distanza

Esempio di applicazione di *k-NN*



- Occorre classificare i punti in *tondi* e *quadrati*
- La stella rappresenta un nuovo punto da classificare
- La risposta dipende dal valore di *k:* se è 3 il punto è classificato come *tondo,* come *quadrato* se è 5

Il Processo del Metodo *k*-NN



Scelta del Parametro *k*

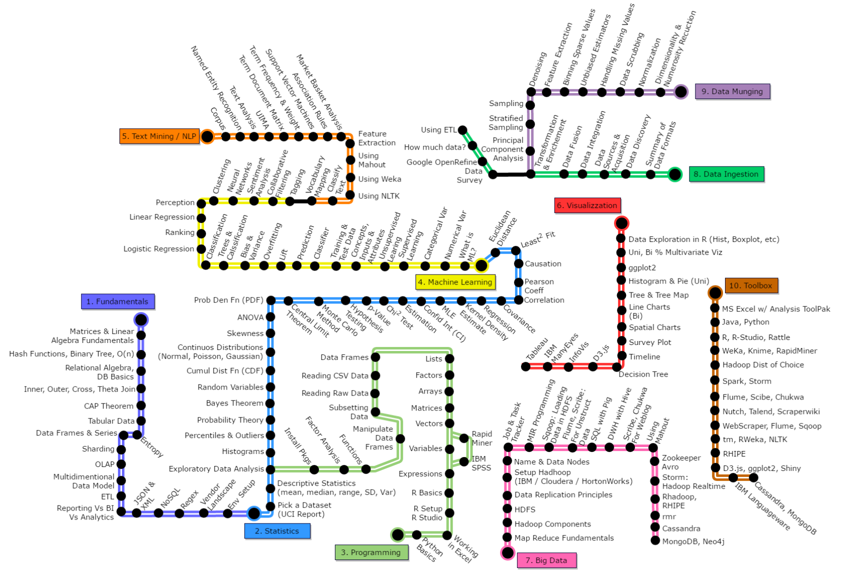
- Il miglior valore dipende dai dati in input
- Valori più grandi di *k* riducono gli effetti distorsivi (errori, valori mancanti) nei dati, ma rendono anche meno marcati i confini tra le classi
- Un valore ottimale viene spesso trovato tramite euristiche
- La Cross Validation è una delle tecniche più frequentemente utilizzate per trovare il valore ottimale per k e per la misura di distanza

Cross Validation

- Tecnica sperimentale per inferire valori ottimali per parametri di modelli predittivi
- Ad esempio, per trovare il valore ottimale di *k* nel *k*-NN o valutare la bontà di un valore di *k*
  - si sceglie un numero *v* in modo casuale e, per ogni valore potenziale di *k,* si classifica, a turno, uno dei *v* insiemi usando gli altri *v-1* come training set
  - per ognuno di tali *v* esperimenti si calcola l'errore quadratico medio, aggregando tali valori
  - si sceglie il valore di *k che* minimizza l'errore totale

Outline

- Introduction to Classification/Regression
  - Training vs test set split
  - Classification/Regression rules
  - Types of classifiers
- **Evaluation of Classifiers**
  - Evaluation criteria
  - Model evaluation
    - Cross-Validation
    - Confusion matrices
    - Accuracy, Precision, Recall, and F1-score, the ROC Curve
- Classification algorithms: Naïve Bayes
- Other types of classification



Metrics for evaluating Classifiers

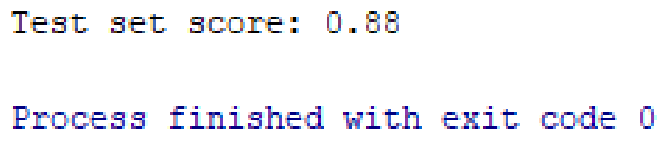
- Classification accuracy
  - Model evaluation and improvement
  - Hit rate
- Classification speed
  - Construction of the classification model;
  - Classification of new objects or cases
- Robustness:
  - Capability of the predictive model to make sufficiently accurate classifications also in presence of errors or missing values
- Scalability
  - Capability to efficiently construct a predictive model even starting from huge volume data
- Easy interpretation of results
  - Trasparency
  - Easy explanation

Model Evaluation and Improvement

- To evaluate supervised models we have to
  - split the dataset into a training set and a test set
    - the `train_test_split` function can be used
  - build a model on the training set
    - the `fit` function can be used
  - evaluate it on the test set
    - For instance, the `score` function can be used
    - It computes the fraction of correctly classified samples

An example of model evaluation

```python
from sklearn.datasets import make_blobs
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

# create a synthetic dataset with data (X) and labels (y)
X, y = make_blobs(random_state=0)
# split data and labels into training and test sets [70-30%]
X_train, X_test, y_train, y_test = train_test_split(X, y,
random_state=0)
# instantiate a model and fit it to the training set
logreg = LogisticRegression().fit(X_train, y_train)
# evaluate the model on the test set
print("Test set score: {:.2f}".format(logreg.score(X_test,
y_test)))
```



```
Test set score: 0.88

Process finished with exit code 0
```

Cross-Validation (1)

- Cross-validation is a stable statistical method of evaluating generalization performances

"The data is split repeatedly and multiple models are trained"

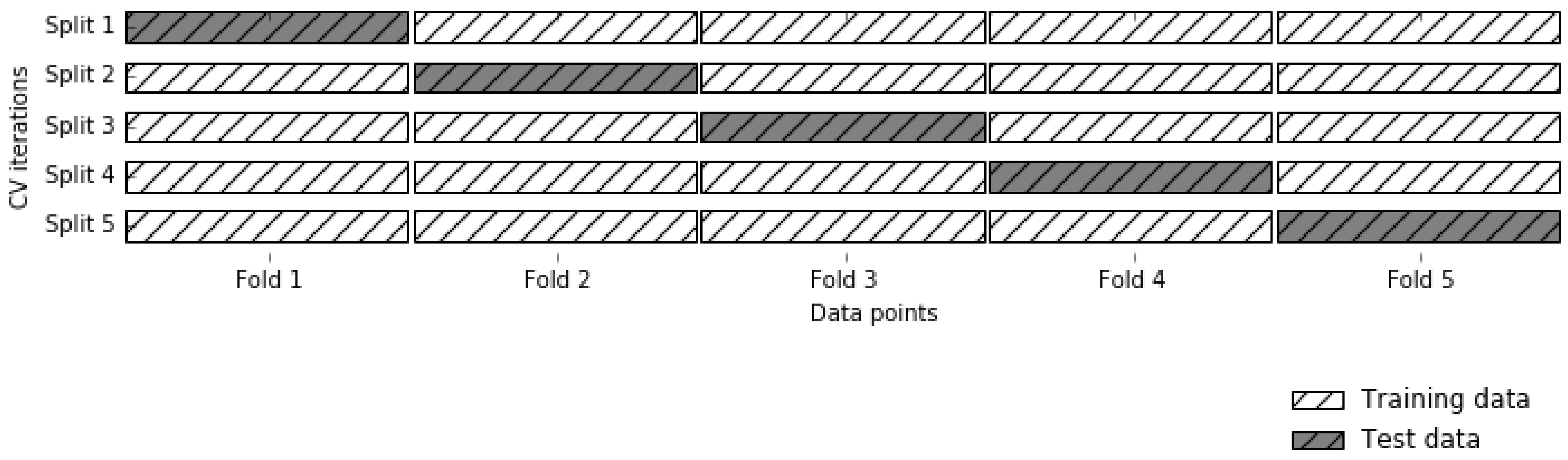
- The most commonly used version of cross-validation is k-fold cross-validation
  - the data is first partitioned into parts of (approximately) equal size, called folds
  - *k* is a user-specified number
  - *k* defines the number of folds

Cross-Validation (2)

- In practice by considering a 5-folds cross validation:
  - The first model is trained using the first fold as test set, and the remaining ones (2–5) as training set
  - The model is built using the data in folds 2–5, and then the accuracy is evaluated on fold 1
  - Then, another model is built using fold 2 as test set and the data in folds 1, 3, 4, and 5 as training set
  - This process is repeated using folds 3, 4, and 5 as test sets
  - The model is evaluated for each of these five splits

Cross-Validation (3)

- The process is illustrated in Figure



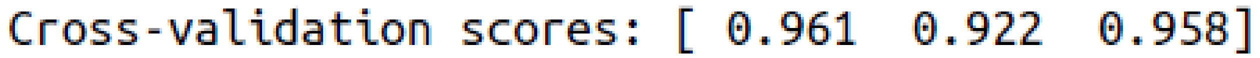
Cross-Validation in scikit-learn (1)

- Cross-validation is implemented in scikit-learn using the `cross_val_score` function from the model_selection module
- The parameters of the `cross_val_score` function are:
  - the model we want to evaluate,
  - the training data, and
  - the ground-truth labels

Cross-Validation in scikit-learn (2)

- Let’s evaluate LogisticRegression on the iris dataset:

```
from sklearn.model_selection import cross_val_score
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression

iris = load_iris()
logreg = LogisticRegression()
scores = cross_val_score(logreg, iris.data, iris.target)
print("Cross-validation scores: {}".format(scores))
```



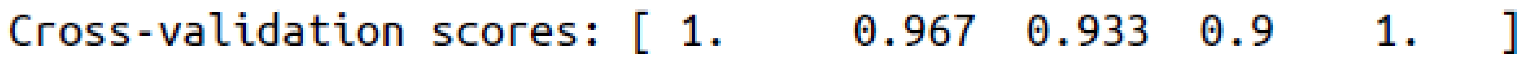
```
Cross-validation scores: [ 0.961  0.922  0.958]
```

- By default, cross_val_score performs three-fold cross-validation, returning three accuracy values.

Cross-Validation in scikit-learn (3)

- We can change the default number of folds used by changing the `cv` parameter:

```python
from sklearn.model_selection import cross_val_score
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
iris = load_iris()

logreg = LogisticRegression()
scores = cross_val_score(logreg, iris.data, iris.target, cv=5)
print("Cross-validation scores: {}".format(scores))
```



```
Cross-validation scores: [ 1.     0.967  0.933  0.9    1.   ]
```

Cross-Validation in scikit-learn (4)

- A common way to summarize the cross-validation accuracy is performed by computing the mean:

```python
from sklearn.model_selection import cross_val_score
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression

iris = load_iris()
logreg = LogisticRegression()
scores = cross_val_score(logreg, iris.data, iris.target, cv=5)

print("Average cross-validation score: {:.2f}".format(scores.mean()))
```

```
Average cross-validation score: 0.96
```

Cross-Validation in scikit-learn (5)

- By looking at the mean score of cross-validation
  - we can conclude that we expect the model to be around 96% accurate on average
- By looking at all 5 scores produced by the five-fold cross-validation
  - we can conclude that there is a relatively high variance in the accuracy between folds, ranging from 100% to 90%
- This could imply that the model is very dependent on the particular folds used for training
  - It could be due to the small size of the dataset

Benefits of Cross-Validation

- There are several benefits to using cross-validation w.r.t a single split into a training and a test set
  - The `train_test_split` performs a random split of the data
  - If we are “lucky”, all examples that are hard to classify end up in the training set
    - The test set will only contain “easy” examples, and our test set accuracy will be unrealistically high
  - If we are “unlucky,” we might have randomly put all the hard-to classify examples in the test set and consequently obtain an unrealistically low score

Benefits of Cross-Validation (2)

- When using cross-validation, each example will be in the test set exactly once
  - The model needs to generalize well to all of the samples in the dataset
    - All the scores (and their mean) must be high
- Having multiple splits of the data also provides some information about how sensitive our model is to the selection of the training dataset

Benefits of Cross-Validation (3)

- Another benefit of cross-validation with respect to a single split is that we use our data more effectively
- When using `train_test_split`
  - we usually use 75% of the data for training and 25% of the data for evaluation
- When using 5-fold cross-validation
  - we can use four-fifths of the data (80%) to fit the model in each iteration
- When using 10-fold cross-validation
  - we can use nine-tenths of the data (90%) to fit the model
- More data will usually result in more accurate models

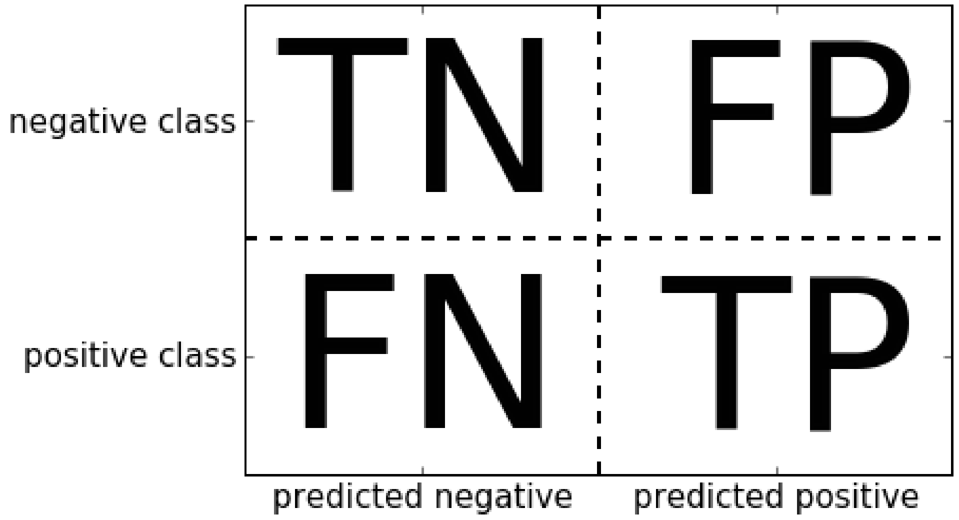
Benefits of Cross-Validation (4)

- The main disadvantage of cross-validation is that it increases computational costs
  - Training *k* models instead of a single model makes the process roughly *k* times slower
- Notice that cross-validation is not a way to build a model that can be applied to new data
  - Cross-validation does not return a model
- When calling `cross_val_score`
  - multiple models are built internally, but
  - the purpose is only to evaluate how well a given algorithm will generalize when trained on a specific dataset

Confusion matrices (1)

One of the most comprehensive ways to represent the evaluation results of classification is using *confusion matrices*

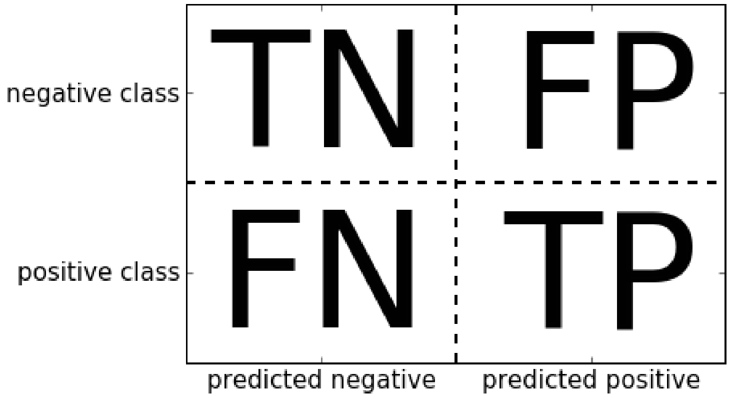
- With binary classifiers a confusion matrix has the form



	predicted negative	predicted positive
negative class	TN	FP
positive class	FN	TP

Confusion matrices (2)

- True positives (TP): the cases for which the classifier correctly predicts a positive case
- True negatives (TN): the cases for which the classifier correctly predicts a negative case

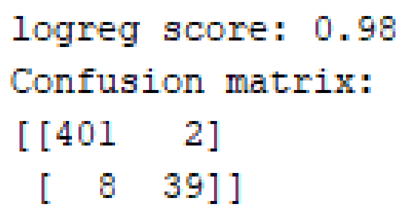


- False positives (FP): the cases for which the classifier incorrectly predicts a negative case as positive
- False negatives (FN): the cases for which the classifier incorrectly predicts a positive case as negative

Confusion matrices in scikit-learn (1)

- Let’s inspect the predictions of LogisticRegression by using the `confusion_matrix` function:

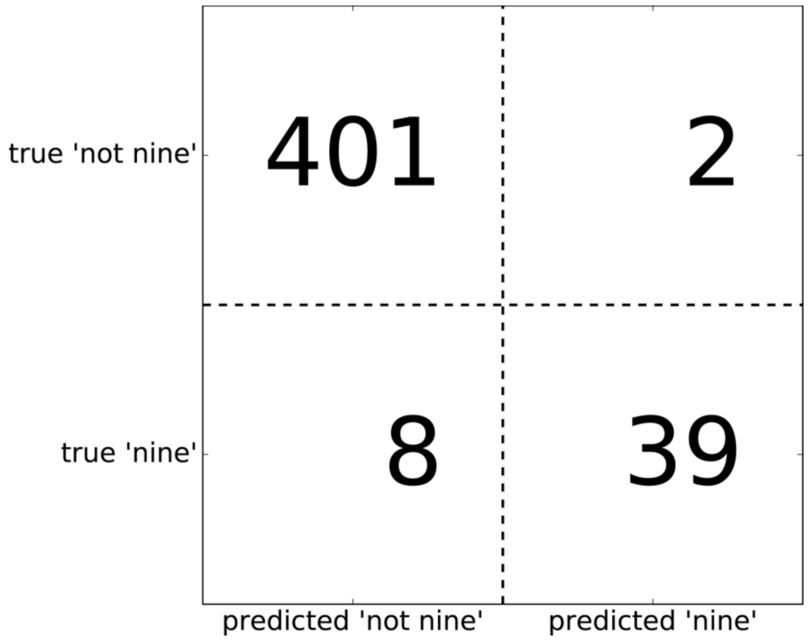
```
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix

digits = load_digits()
y = digits.target == 9
X_train, X_test, y_train, y_test = train_test_split(digits.data,
y, random_state=0)

logreg = LogisticRegression().fit(X_train, y_train)
pred_logreg = logreg.predict(X_test)
print("logreg score: {:.2f}".format(logreg.score(X_test, y_test)))

confusion = confusion_matrix(y_test, pred_logreg)
print("Confusion matrix:\n{}".format(confusion))
```



```
logreg score: 0.98
Confusion matrix:
[[401   2]
 [  8  39]]
```

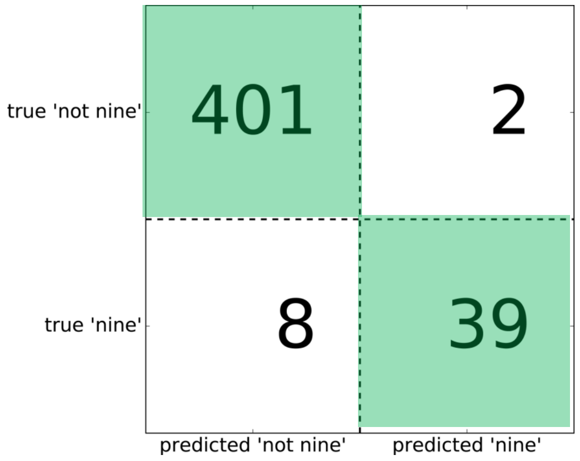
Confusion matrices in scikit-learn (2)

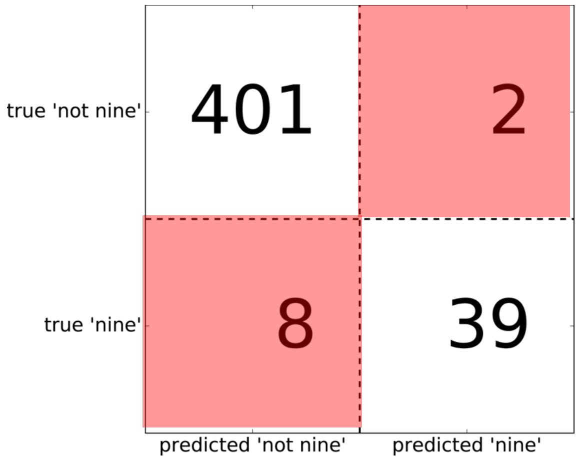
- The output of `confusion_matrix` is a two-by-two array
  - the rows correspond to the true classes, and
  - the columns correspond to the predicted classes
- Each entry counts how often a sample that belongs to the class corresponding to the row was classified as the class corresponding to the column
  - here, “not nine” and “nine”


true 'not nine'
true 'nine'
401
2
8
39
predicted 'not nine'
predicted 'nine'

Confusion matrices results

- Entries on the main diagonal of the confusion matrix correspond to correct classifications





- other entries tell us how many samples of one class got mistakenly classified as another class

Confusion Matrix vs Performance Metrics

		True Class	
		Positive	Negative
Predicted Class	Positive	True Positive Count (TP)	False Positive Count (FP)
	Negative	False Negative Count (FN)	True Negative Count (TN)

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$True\ Positive\ Rate = \frac{TP}{TP + FN}$$

$$True\ Negative\ Rate = \frac{TN}{TN + FP}$$

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

Relation to Accuracy

- One way to summarize the result in the confusion matrix is by computing accuracy
  - Accuracy can be expressed as:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

- The accuracy is the number of correct predictions (TP and TN) divided by the number of all samples (all entries of the confusion matrix summed up)
- In general, there are several other ways to summarize the confusion matrices

Precision

Precision measures how many of the samples predicted as positive are actually positive

$$Precision = \frac{TP}{TP + FP}$$

- Precision is used as a performance metric when the goal is to limit the number of false positives
- Precision is also known as positive predictive value (PPV)

Precision: An example

- Imagine a model for predicting whether a new drug will be effective in treating a disease in clinical trials
  - Clinical trials are notoriously expensive, and
  - A pharmaceutical company will only want to run an experiment if it is very sure that the drug will actually work
- In this case, it is important that the model does not produce many false positives
- In other words, that it has a high precision

Recall

Recall measures how many of the positive samples are captured by the positive predictions

$$Recall = \frac{TP}{TP + FN}$$

- Recall is used as performance metric when we need to identify all positive samples
  - when it is important to avoid false negatives
- Other names for recall are sensitivity, hit rate, or true positive rate (TPR)

Recall: An example

- Imagine a model for predicting the cancer diagnosis
  - It is important to find all people that are sick, possibly including healthy patients in the prediction
- In this scenario it is clear that
  - we want to avoid false negatives as much as possible
  - false positives can be viewed as more of a minor nuisance

Precision/Recall Tradeoff

- Unfortunately, increasing precision reduces recall, and vice versa
  - This is called the precision/recall tradeoff
- Predicting all samples as positive will result in many false positives
  - The recall will be perfect - The precision will be very low
- On the other hand, a model that predicts only the single data point it is most sure about as positive and the rest as negative
  - The precision will be perfect - The recall will be very low

F_1-score

- While precision and recall are very important measures, looking at only one of them will not provide you with the full picture
- One way to summarize them is the f-score or f-measure, which is with the harmonic mean of precision and recall:

$$F_1 = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}} = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

F_1-score in scikit-learn

- Let’s run it on the predictions for the “nine vs. rest” dataset
  - we will assume that the “nine” class is the positive class

```
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score

digits = load_digits()
y = digits.target == 9
X_train, X_test, y_train, y_test =
train_test_split(digits.data, y, random_state=0)
logreg = LogisticRegression().fit(X_train, y_train)
pred_logreg = logreg.predict(X_test)

print("logreg score: {:.2f}".format(logreg.score(X_test,
y_test)))
print("f1 score logistic regression:
{:.2f}".format(f1_score(y_test, pred_logreg)))
```

```
logreg score: 0.98
f1 score logistic regression: 0.89

Process finished with exit code 0
```

Precision, Recall, F_1-score in scikit-learn

- For a more comprehensive summary of precision, recall, and f1-score, we can use the `classification_report` convenience function

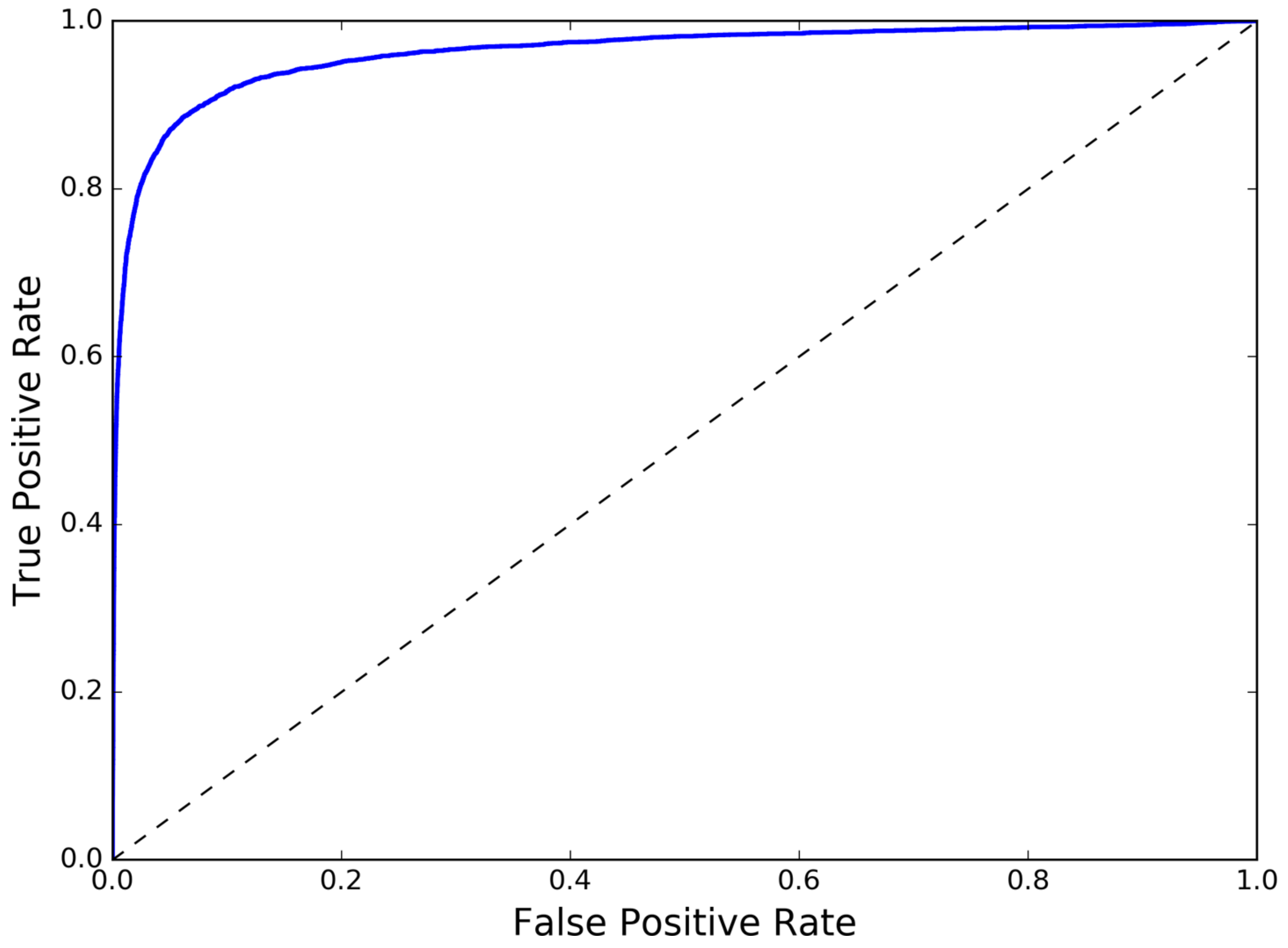
```
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

digits = load_digits()
y = digits.target == 9
X_train, X_test, y_train, y_test = train_test_split(digits.data, y, random_state=0)
logreg = LogisticRegression().fit(X_train, y_train)
pred_logreg = logreg.predict(X_test)

print(classification_report(y_test, pred_logreg, target_names=["not nine", "nine"]))
```

```
              precision    recall  f1-score   support

    not nine       0.98      1.00      0.99       403
        nine       0.95      0.83      0.89        47

   micro avg       0.98      0.98      0.98       450
   macro avg       0.97      0.91      0.94       450
weighted avg       0.98      0.98      0.98       450
```

The ROC Curve (1)

- The Receiver Operating Characteristic (ROC) curve is another common tool used with binary classifiers
- It is very similar to the precision/recall curve, but instead of plotting precision versus recall, the ROC curve plots the *true positive rate* (recall) against the false positive rate (FPR)
- The FPR is the ratio of negative instances that are incorrectly classified as positive
    - It is equal to one minus the *true negative rate*
    - The TNR is also called *specificity*

The ROC Curve (1)



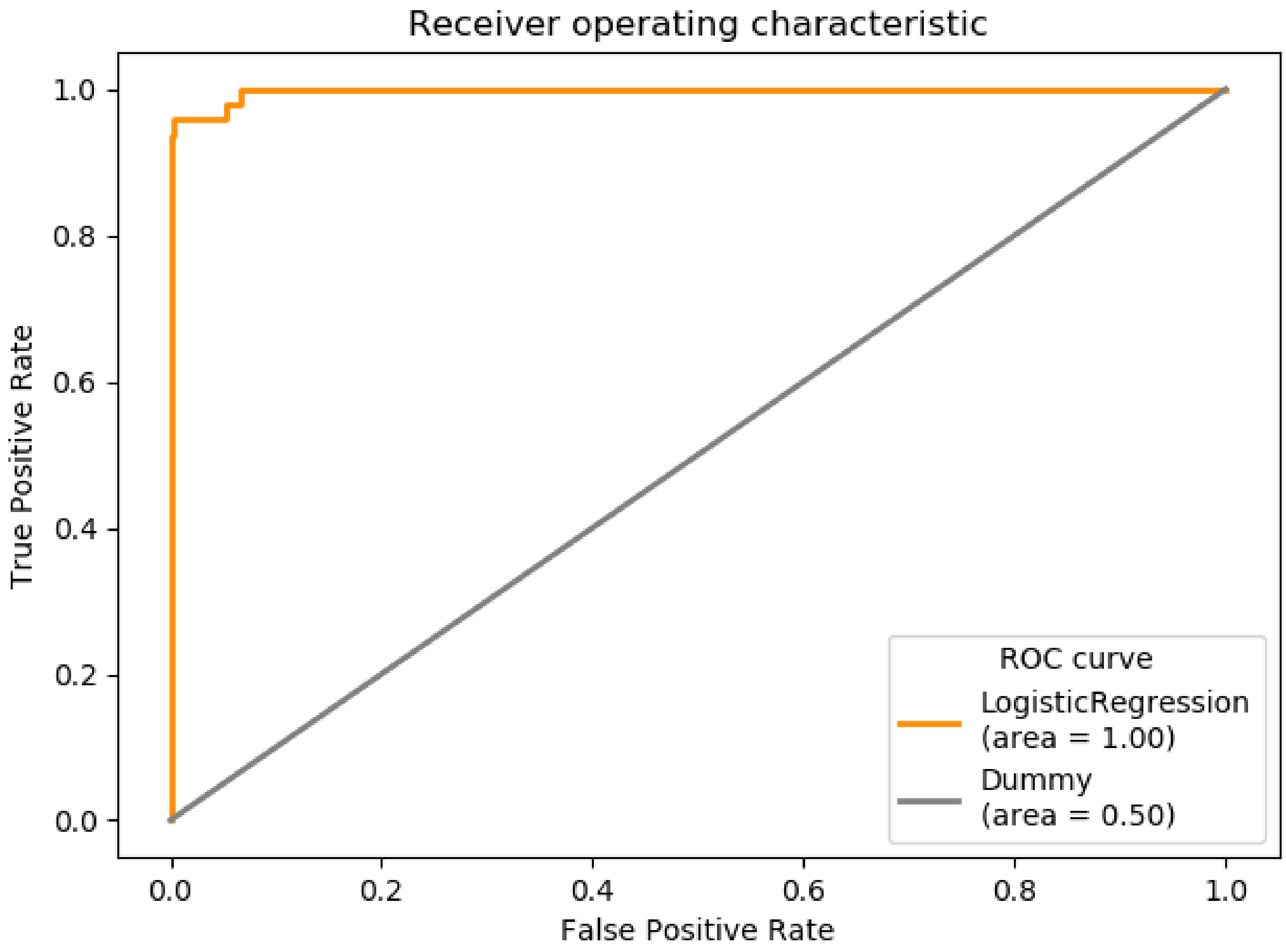
ROC Curve Tradeoff

- Once again there is a tradeoff
  - the higher the recall (TPR), the more false positives (FPR) the classifier produces
- The dotted line in the previous Figure represents the ROC curve of a purely random classifier
- A good classifier stays as far away from that line as possible
  - toward the top-left corner

ROC Curve in scikit-learn

```
from sklearn import datasets, metrics, model_selection, dummy
from sklearn.linear_model import LogisticRegression
import matplotlib.pyplot as plt # To plot the Curve
digits = datasets.load_digits()
sevens = (digits.target == 7)
classifier = LogisticRegression()
most_frequent = dummy.DummyClassifier(strategy='most_frequent')
X_train, X_test, y_train, y_test = model_selection.train_test_split(digits.data,
sevens, random_state=0)
y_score = classifier.fit(X_train, y_train).predict_proba(X_test)
most_frequent_score = most_frequent.fit(X_train, y_train).predict_proba(X_test)
fpr, tpr, _ = metrics.roc_curve(y_test, y_score[:, 1])
roc_auc = metrics.auc(fpr, tpr)
fpr_dummy, tpr_dummy, _ = metrics.roc_curve(y_test, most_frequent_score[:, 1])
roc_auc_dummy = metrics.auc(fpr_dummy, tpr_dummy)
lw = 2
plt.plot(fpr, tpr, color='darkorange', lw=lw,
label='LogisticRegression\n(area = %0.2f)' % roc_auc)
plt.plot(fpr_dummy, tpr_dummy, color='.5', lw=lw,
label='Dummy\n(area = %0.2f)' % roc_auc_dummy)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic')
plt.legend(loc="lower right", title='ROC curve')
plt.tight_layout()
plt.show()
```

ROC Curve in scikit-learn

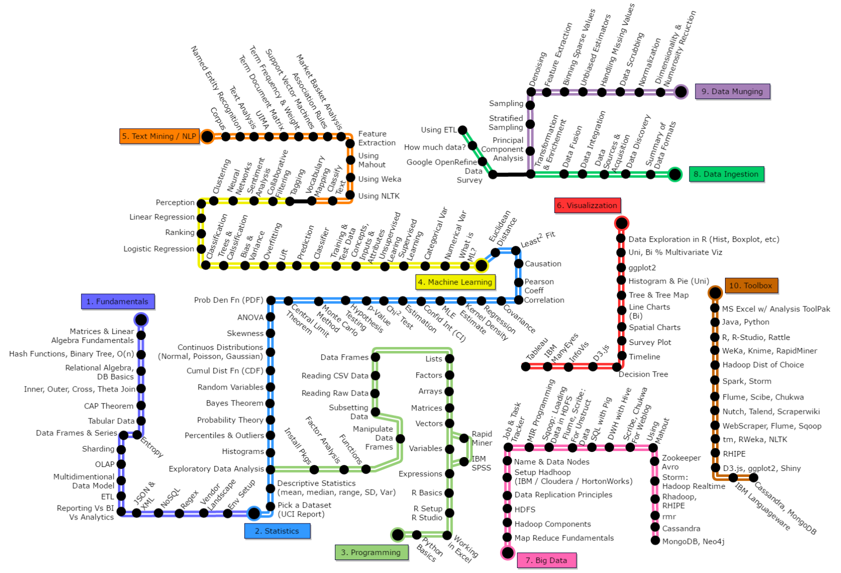


ROC Curve / PR Curve

- Since the ROC curve is so similar to the precision/recall (or PR) curve, you may wonder how to decide which one to use
- you should prefer the PR curve when
  - the positive class is rare,
  - you care more about the false positives than the false negatives
- the ROC curve otherwise

Outline

- Introduction to Classification/Regression
  - Training vs test set split
  - Classification/Regression rules
  - Types of classifiers
- Evaluation of Classifiers
  - Evaluation criteria
  - Model evaluation
    - Cross-Validation
    - Confusion matrices
    - Accuracy, Precision, Recall, and F1-score, the ROC Curve
- **Classification algorithms: Naïve Bayes**
- Other types of classification



A Probabilistic Classifier

- The Naïve Bayes classifier belongs to the family of probabilistic classifiers.
- For each output class, it computes the probability of each predictive attribute (feature) value for the data belonging to that class, so as to predict the pobability distribution over all output classes.
- Naturally, from the resulting probability distribution we can derive the class to which a new sample of data most likely belongs.

Naïve Bayes in practice

- As its name indicates, the Naïve Bayes behaves as follows:
  - Bayes: maps the probability of the input features observed for one of the output class to the probability of that class on observed inputs according to the Bayes theorem.
  - Naïve: it simplifies the computation of probability by assuming that the predictive attributes are statistically indipendent.

Bayes Theorem

- Before explaining the Bayes classifier it is important to explain the Bayes Theorem.
- Given two events *A* e *B*.
- The events can be *tomorrow rains*, *two kings are drawn from a deck of playing cards* or that *a person has or not cancer*.
- In the Bayes theorem, $P(A|B)$ is the probability that A occurs given that B is *True*. It can be computed as follows:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Bayes Theorem (2)

- The probability that A occurs given that B is *True* can be computed as follows:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

- $P(B|A)$ is the probability that B occurs when A is True
- $P(A)$ and $P(B)$ are the probabilities that A and B occur, respectively.

Bayes Theorem: Example

- Given two coins,
  - One is *Unfair*, with 90% of tosses yielding *Head* and 10% *Tail*
  - Whereas the other one is *Fair*
- Choose one of the two coins randomly and toss it:

  What is the probability that such coin is the Unfair one, if the outcome is Head?

Bayes Theorem: Example(2)

- The probability to have chosen the *Unfair* coin when the outcome is *Head,* $P(U|H)$, can be computed as follows:

$$P(U|H) = \frac{P(H|U)P(U)}{P(H)}$$

- We now that:
  - $P(H|U)$ is 90%.
  - $P(U)$ is 0,5 because we randomly chose one coin out of two.
- How can we compute the probability $P(H)$ to get a *Head*?

Bayes Theorem: Example(2)

- Computing the probability to get a *Head*, $P(H)$, is not simple, because it depends on 2 independent events,

$$P(H) = P(H|U)P(U) + P(H|F)P(F)$$

  - where U is when the *Unfair* coin is chosen and F is when the *Fair* one is chosen

- Thus, $P(U|H)$ can be computed as follows:

$$P(U|H) = \frac{P(H\,|\,U)P(U)}{P(H)} = \frac{P(H\,|\,U)P(U)}{P(H\,|\,U)P(U)+P(H\,|\,F)P(F)} = \frac{0.9 * 0.5}{0.9 * 0.5 + 0.5 * 0.5} = 0.64$$

Bayes Theorem: Example2

- Suppose a doctor has reported the following results in a cancer screening test over 10.000 persons:

	Cancer	No Cancer	Total
Test Positive	80	900	980
Test Negative	20	9000	9020
Total	100	9900	10000

- This means that:
  - 80 oncological patients out of 100 have been correctly diagnosed, whereas the remaining 20 ones not.
  - Cancer is erroneusly reported in 900 healthy persons out of 9.900.

Bayes Theorem: Example2(2)

- If a patient tests positive on this cancer test, what is the probability that s/he really has cancer?
- Let:
  - *C* be the event of having cancer
  - *Pos* the event to test positive on the cancer test
- We have:

$$P(C|Pos) = \frac{P(Pos|C)P(C)}{P(Pos)}$$

Bayes Theorem: Example2(3)

- The probability to have cancer when testing positive to the test, $P(C|Pos)$, can be computed as follows:

$$P(C|Pos) = \frac{P(Pos|C)P(C)}{P(Pos)} = \frac{0.8 * 0.01}{0.098} = 8.16\ \%$$

- where:
  - $P(Pos|C) = \frac{80}{100} = 0.8$
  - $P(C) = \frac{100}{10000} = 0.01$
  - $P(Pos) = \frac{980}{10000} = 0.098$

The Naïve Bayes Classifier

- Let us see how the Naïve Bayes algorithm works.
- Given a training instance x=(x_1,x_2, …,x_n) with *n* features, the goal of Naïve Bayes is to determine the probability for x to belong to each of the *K* classes y_1,y_2, …,y_K
  - We must consider a joint event that considers the values of the observed features x=(x_1,x_2, …,x_n)
  - Then, we must consider the probability $P(y_k|x)$ that the instance x belongs to the class k.

Let us apply Bayes Theorem

- We can immmediately apply the Bayes Theorem:

$$P(y_k|x) = \frac{P(x|y_k)P(y_k)}{P(x)}$$

- Where:
  - $P(y_k)$ indicates how the *k classes are distributed, without further knowledge of the characteristics of the observation.*
    - In the Bayesian probability terminology it is also called *Prior*.
    - The Prior can be predetermined (usually in a uniform way, where each class has the same probability to occur) or it is learned from a set of training data.

Posterior and Likelihood

$$P(y_k|x) = \frac{P(x|y_k)P(y_k)}{P(x)}$$

- $P(y_k|x)$ is said posterior, and it possesses an additional knowledge given by the observation
- $P(x|y_k)$ is said likelihood, and it represents the joint distribution of n features considering that the instance x belongs to the class y_k
- $P(x)$ is said evidence and it depends exclusively from the overall distribution of the features, hence it represents normalization constant.

Computation of the Likelihood

- It will be difficult to compute the likelihood when the number of features increases
- In Naïve Bayes this problem is solved thanks to the naïve hypothesis of independence of the features:

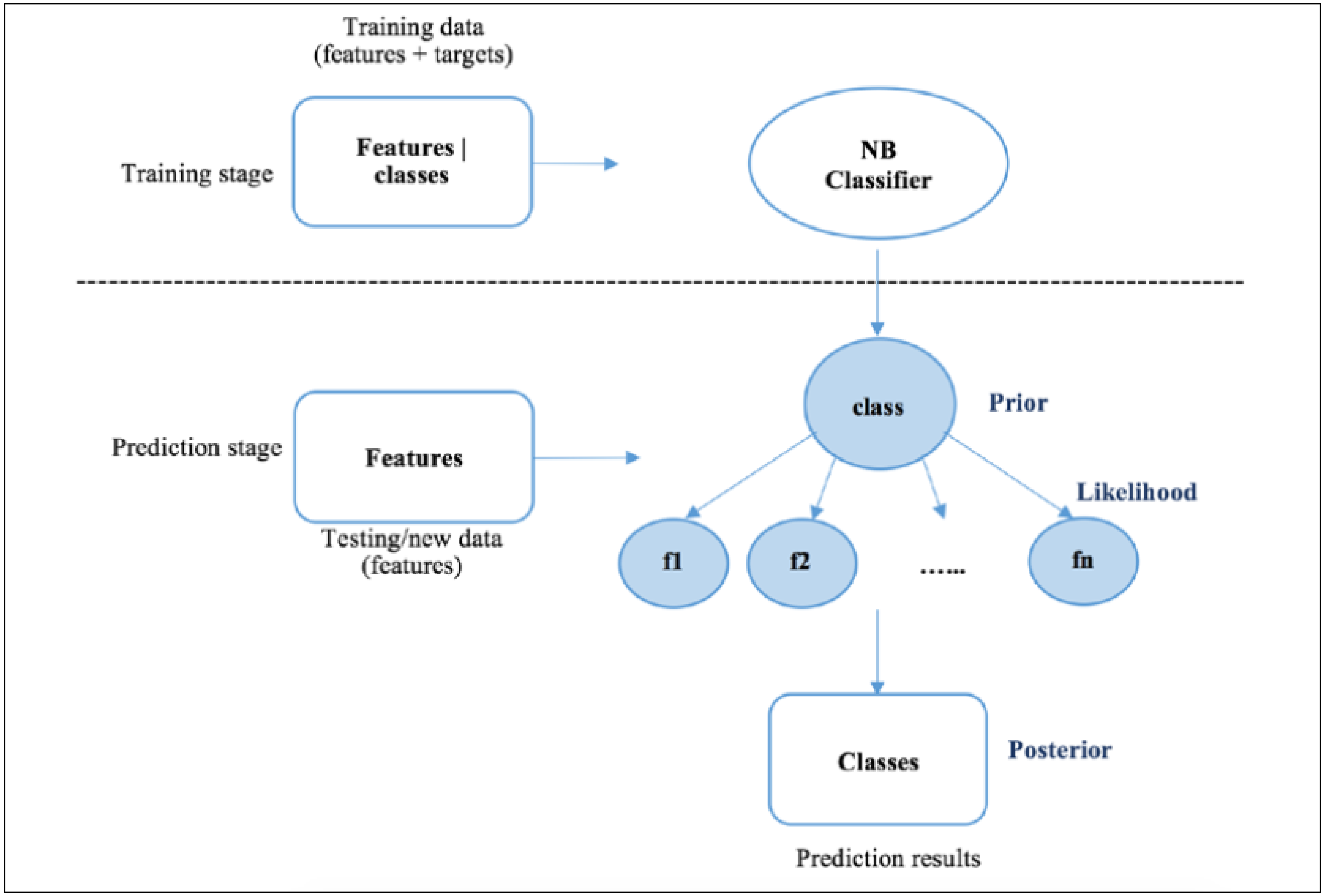
$$P(x|y_k) = P(x_1|y_k) * P(x_2|y_k) * \dots * P(x_n|y_k)$$

- Thus, *Bayes* theorem says that the *posterior* is proportional to the *likelihood* and the *prior*

$$P(y_k|x) \propto P(x|y_k)\, P(y_k)$$

$$P(x|y_k)P(y_k) = P(x_1|y_k) * P(x_2|y_k) * \dots * P(x_n|y_k) * P(y_k)$$

Schema of Naïve Bayes Classificator

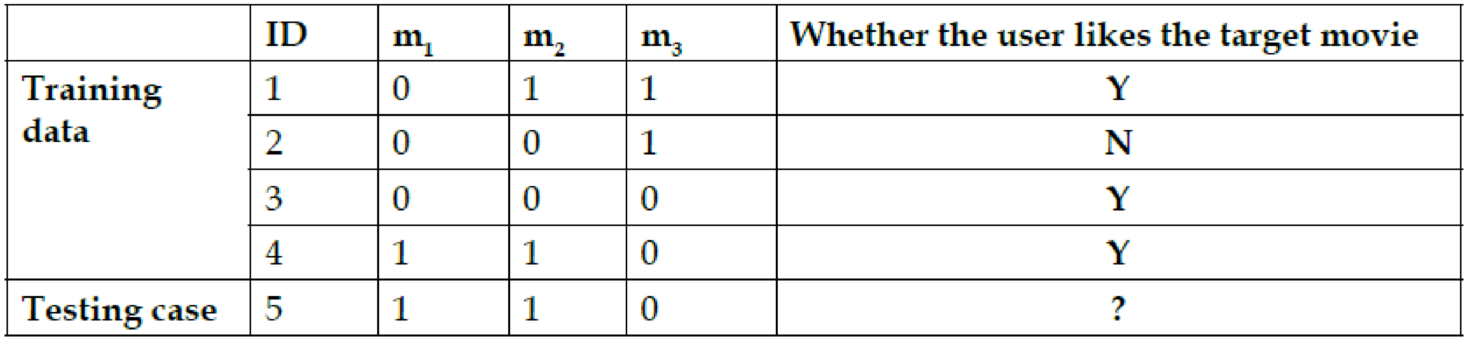


Naïve Bayes Classificator: Example

- Given four (pseudo) users, if each of them
  - doesn't like/likes three movies m_1,m_2, m_3 (indicated as 0 or 1, respectively)
  - doesn't like/likes a target movie (denoted as N o Y, respectively)
- What's the probability that a fifth user likes the target movie?

	ID	m_1	m_2	m_3	Whether the user likes the target movie
Training data	1	0	1	1	Y
	2	0	0	1	N
	3	0	0	0	Y
	4	1	1	0	Y
Testing case	5	1	1	0	?

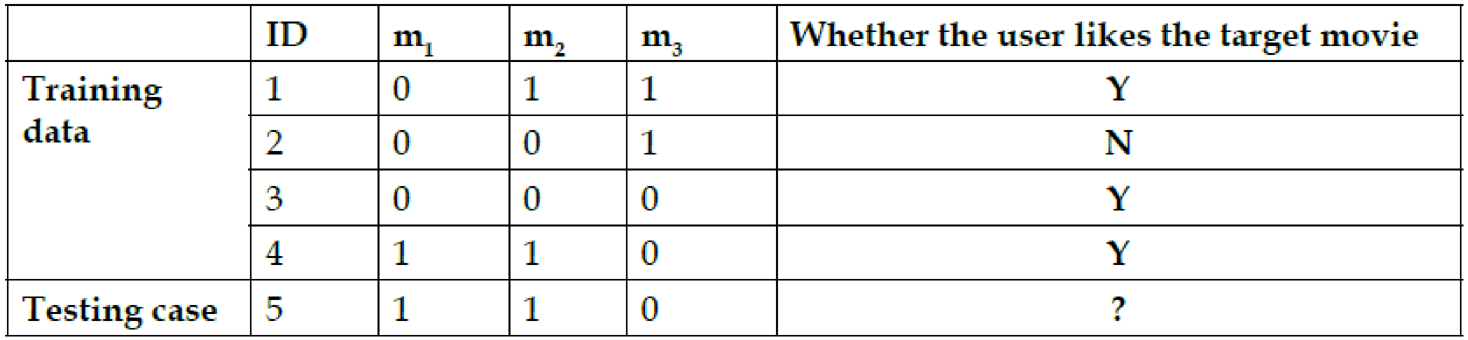
Naïve Bayes Classificator: Example



	ID	m_1	m_2	m_3	Whether the user likes the target movie
Training data	1	0	1	1	Y
	2	0	0	1	N
	3	0	0	0	Y
	4	1	1	0	Y
Testing case	5	1	1	0	?

- In this scenario
  - m_1,m_2, m_3 represent the features (Predictive attributes)
  - there are 4 training samples, also containing the target value
  - from the training set we know that $P(Y)$ $=$ 3/4 and $P(N)$ $=$ 1/4
  - for sake of simplicity let us denote with f_1,f_2, and f_3, respectively, the events that a user likes the 3 movies

Naïve Bayes Classificator: Example



	ID	m_1	m_2	m_3	Whether the user likes the target movie
Training data	1	0	1	1	Y
	2	0	0	1	N
	3	0	0	0	Y
	4	1	1	0	Y
Testing case	5	1	1	0	?

- To compute $P(Y|x)$, with x=(1,1,0), first we need to compute the likelihoods:
  - $P(f_1 = 1|Y), P(f_2 = 1|Y)$ and $P(f_3 = 0|Y)$
  - Similarly, we should compute $P(f_1 = 1|N), P(f_2 = 1|N)$ and $P(f_3 = 0|N)$ based on the values of the training set
  - When the event is not observable on the training set, Laplace smoothing is applied to avoid that result reduces to 0.

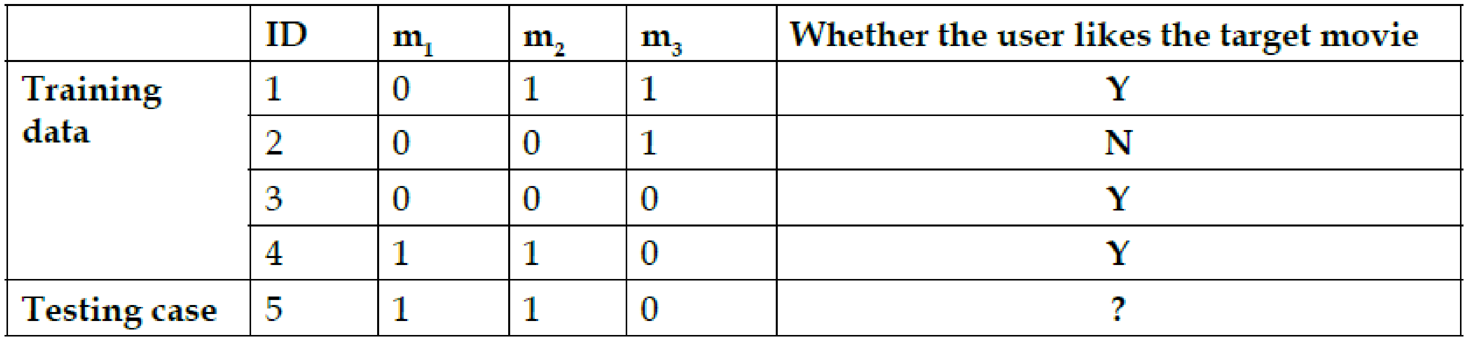
Naïve Bayes Classificator: Example

	ID	m_1	m_2	m_3	Whether the user likes the target movie
Training data	1	0	1	1	Y
	2	0	0	1	N
	3	0	0	0	Y
	4	1	1	0	Y
Testing case	5	1	1	0	?

- $P(f_1 = 1|N) = \dfrac{\#instances\ liking\ m_1\ but\ not\ the\ target}{\#instances\ not\ liking\ the\ target} = \dfrac{0}{1}$
  - since only 1 training instance $(\mathrm{ID} = 2)$ $doesn't\ like$ the target, but it has $f_1 = 0$ (*doesn't like* m_1), hence numerator is *0* and denominator 1
  - since the ratio reduces to 0, we apply Laplace smoothing by adding 1 to the numerator, and (1*2) to the denominator, since the number of possible target values is 2 (*Y* or *N*):

$$P(f_1 = 1|N) = \frac{0+1}{1+2} = \frac{1}{3}$$

Naïve Bayes Classificator: Example



	ID	m_1	m_2	m_3	Whether the user likes the target movie
Training data	1	0	1	1	Y
	2	0	0	1	N
	3	0	0	0	Y
	4	1	1	0	Y
Testing case	5	1	1	0	?

- $P(f_1 = 1|Y) = \dfrac{\#instances\ liking\ m_1\ and\ the\ target}{\#instances\ liking\ the\ target} = \dfrac{1}{3}$
  - since 3 training instances $(\mathrm{ID} = 1{,}3{,}4)$ *like* the target, but only one of them $(\mathrm{ID} = 4)$ has $f_1 = 1$ (*likes* m_1), numerator is 1 and denominator 3
  - we still apply Laplace smoothing because the result is close to zero:

$$P(f_1 = 1|Y) = \frac{1+1}{3+2} = \frac{2}{5}$$

Naïve Bayes Classificator: Example

	ID	m_1	m_2	m_3	Whether the user likes the target movie
Training data	1	0	1	1	Y
	2	0	0	1	N
	3	0	0	0	Y
	4	1	1	0	Y
Testing case	5	1	1	0	?

- $P(f_1 = 1|N) = \frac{0+1}{1+2} = \frac{1}{3}$ and $P(f_1 = 1|Y) = \frac{1+1}{3+2} = \frac{2}{5}$
- $P(f_2 = 1|N) = \frac{0+1}{1+2} = \frac{1}{3}$ and $P(f_2 = 1|Y) = \frac{2+1}{3+2} = \frac{3}{5}$
- $P(f_3 = 1|N) = \frac{0+1}{1+2} = \frac{1}{3}$ and $P(f_3 = 1|Y) = \frac{2+1}{3+2} = \frac{3}{5}$

Naïve Bayes Classificator: Example

- Now we can compute the two Posterior values:

$$P(N|x) \propto P(N)*P(f_1 = 1|N)*P(f_2 = 1|N)*P(f_3 = 0|N) = \frac{1}{4}*\frac{1}{3}*\frac{1}{3}*\frac{1}{3} = \frac{1}{108} = 0.0092$$

$$P(Y|x) \propto P(Y)*P(f_1 = 1|Y)*P(f_2 = 1|Y)*P(f_3 = 0|Y) = \frac{3}{4}*\frac{2}{5}*\frac{3}{5}*\frac{3}{5} = \frac{54}{500} = 0.108$$

- Since this is a probability distribution, let us normalize these values so that they sum to *1*:

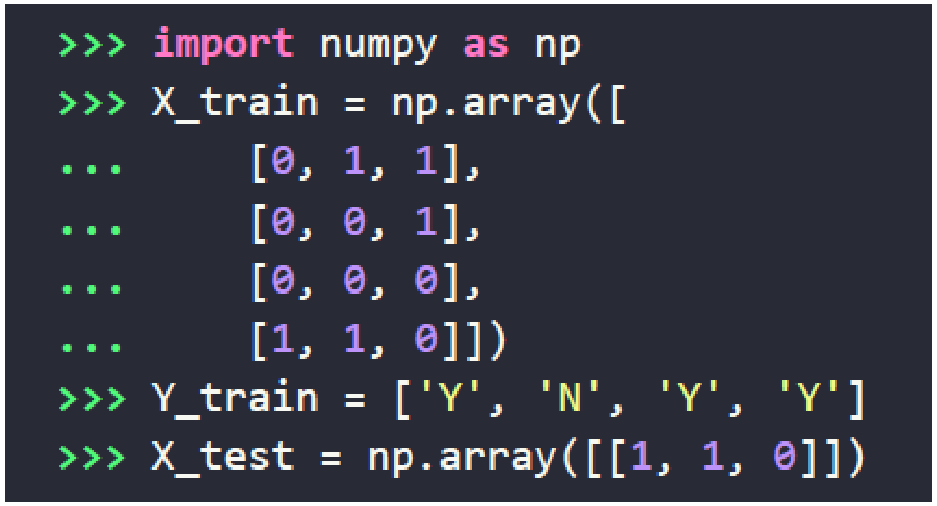
$$P(N|x) = \frac{0,0092}{0.0092+0.108} = 0.078$$

$$P(Y|x) = \frac{0.108}{0,0092+0.108} = 0.922$$

- Probability the new user likes the target movie is 92,2%!

Implementing a Naïve Bayes Classifier

- Before developing the model, let us define a syntetic dataset to work with



```
>>> import numpy as np
>>> X_train = np.array([
...     [0, 1, 1],
...     [0, 0, 1],
...     [0, 0, 0],
...     [1, 1, 0]])
>>> Y_train = ['Y', 'N', 'Y', 'Y']
>>> X_test = np.array([[1, 1, 0]])
```

- It is the same we have seen in the example.

Implementing a Naïve Bayes Classifier

- Let us define a function grouping samples based on their target value

```
>>> def get_label_indices(labels):
...     """
...     Group samples based on their labels and return indices
...     @param labels: list of labels
...     @return: dict, {class1: [indices], class2: [indices]}
...     """
...     from collections import defaultdict
...     label_indices = defaultdict(list)
...     for index, label in enumerate(labels):
...         label_indices[label].append(index)
...     return label_indices
```

```
>>> label_indices = get_label_indices(Y_train)
>>> print('label_indices:\n', label_indices)
label_indices
 defaultdict(<class 'list'>, {'Y': [0, 2, 3], 'N': [1]})
```

Computing the *Prior*

- Let us define a function to compute the *Prior* $P(y_k)$ considering label_indices

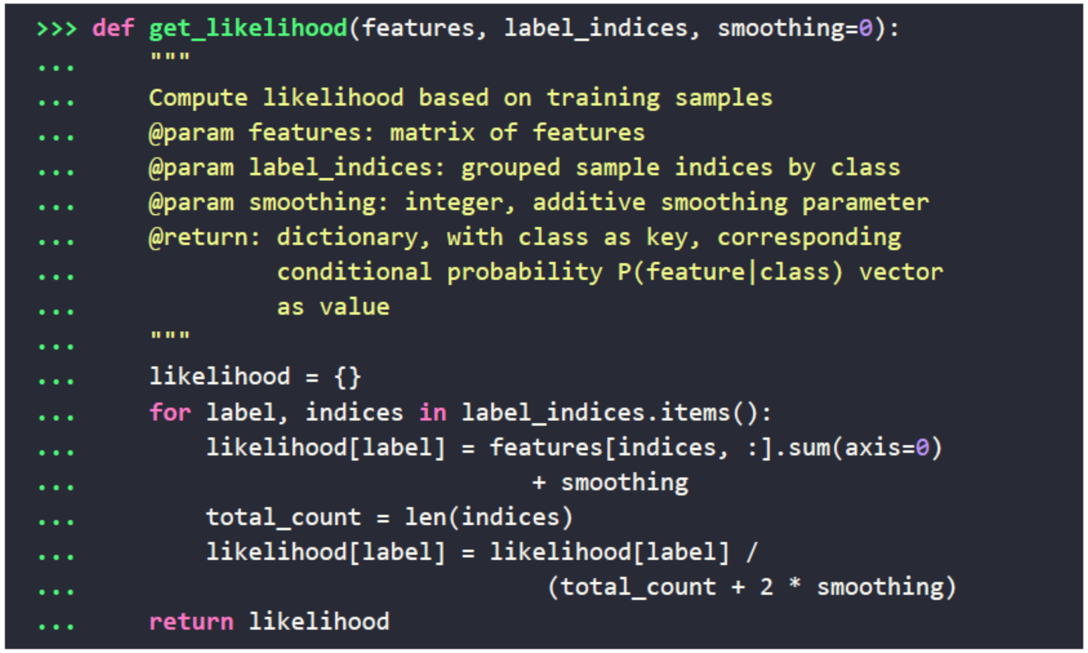
```
>>> def get_prior(label_indices):
...     """
...     Compute prior based on training samples
...     @param label_indices: grouped sample indices by class
...     @return: dictionary, with class label as key, corresponding
...              prior as the value
...     """
...     prior = {label: len(indices) for label, indices in
...                                       label_indices.items()}
...     total_count = sum(prior.values())
...     for label in prior:
...         prior[label] /= total_count
...     return prior
```

```
>>> prior = get_prior(label_indices)
>>> print('Prior:', prior)
 Prior: {'Y': 0.75, 'N': 0.25}
```

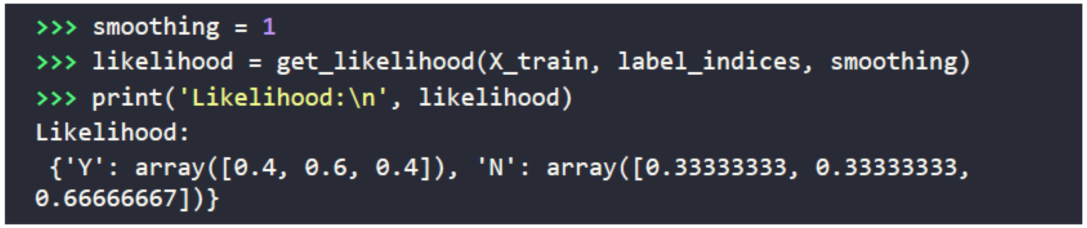
Computing the *Likelyhood*

- Let us define a function to compute the *Likelihood* $P(feature|class)$ based on the previously computed prior values

- In the example we set the smoothing parameter to 1. We could set it to 0 so as not to apply it, or any other positive value.



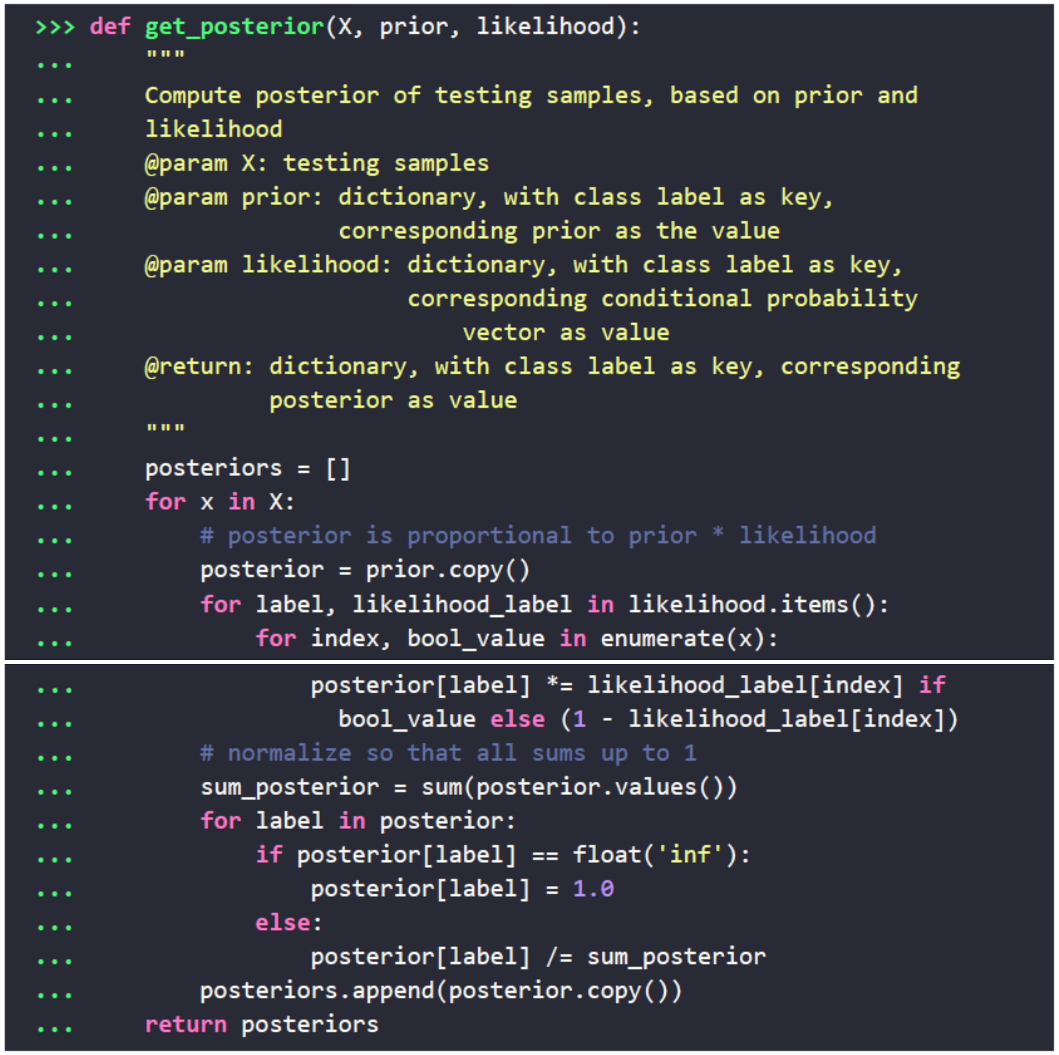
```
>>> def get_likelihood(features, label_indices, smoothing=0):
...     """
...     Compute likelihood based on training samples
...     @param features: matrix of features
...     @param label_indices: grouped sample indices by class
...     @param smoothing: integer, additive smoothing parameter
...     @return: dictionary, with class as key, corresponding
...              conditional probability P(feature|class) vector
...              as value
...     """
...     likelihood = {}
...     for label, indices in label_indices.items():
...         likelihood[label] = features[indices, :].sum(axis=0)
...                                 + smoothing
...         total_count = len(indices)
...         likelihood[label] = likelihood[label] /
...                                 (total_count + 2 * smoothing)
...     return likelihood
```



```
>>> smoothing = 1
>>> likelihood = get_likelihood(X_train, label_indices, smoothing)
>>> print('Likelihood:\n', likelihood)
Likelihood:
 {'Y': array([0.4, 0.6, 0.4]), 'N': array([0.33333333, 0.33333333,
0.66666667])}
```

Computing the *Posterior*

- With the *priors* and the *likelihoods* ready, we can now compute the *posterior* for test/new samples



```
>>> def get_posterior(X, prior, likelihood):
...     """
...     Compute posterior of testing samples, based on prior and
...     likelihood
...     @param X: testing samples
...     @param prior: dictionary, with class label as key,
...                   corresponding prior as the value
...     @param likelihood: dictionary, with class label as key,
...                        corresponding conditional probability
...                            vector as value
...     @return: dictionary, with class label as key, corresponding
...              posterior as value
...     """
...     posteriors = []
...     for x in X:
...         # posterior is proportional to prior * likelihood
...         posterior = prior.copy()
...         for label, likelihood_label in likelihood.items():
...             for index, bool_value in enumerate(x):
...                 posterior[label] *= likelihood_label[index] if
...                   bool_value else (1 - likelihood_label[index])
...         # normalize so that all sums up to 1
...         sum_posterior = sum(posterior.values())
...         for label in posterior:
...             if posterior[label] == float('inf'):
...                 posterior[label] = 1.0
...             else:
...                 posterior[label] /= sum_posterior
...         posteriors.append(posterior.copy())
...     return posteriors
```

Scikit Learn BernoulliNB Module

- If we want to avoit writing such code from scratch, we can use the shortcut provided by the Scikit Learn BernoulliNB module

```
>>> from sklearn.naive_bayes import BernoulliNB
```

Implementing Naïve Bayes with Scikit Learn

- Let us initialize the model
  - with a *Smoothing* factor (alpha parameter in scikit-learn) of 1,0, and
  - a *Prior* trained from the training set (specified as fit_prior=True in scikit-learn)

```
>>> clf = BernoulliNB(alpha=1.0, fit_prior=True)
```

- To train the Naïve Bayes classifier with the fit method we use the following line of code

```
>>> clf.fit(X_train, Y_train)
```

Naïve Bayes predictions with Scikit Learn

- Predictions on test/new samples can be easily made through the *predict_proba* method

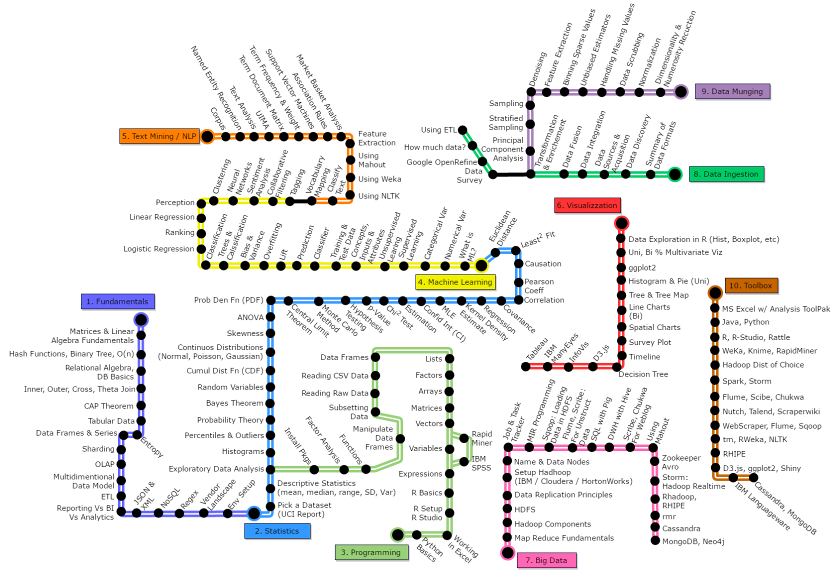
```
>>> pred_prob = clf.predict_proba(X_test)
>>> print('[scikit-learn] Predicted probabilities:\n', pred_prob)
[scikit-learn] Predicted probabilities:
 [[0.07896399 0.92103601]]
```

- To directly obtain the predicted class the *predict* method can be called

```
>>> pred = clf.predict(X_test)
>>> print('[scikit-learn] Prediction:', pred)
[scikit-learn] Prediction: ['Y']
```

Outline

- Introduction to Classification/Regression
  - Training vs test set split
  - Classification/Regression rules
  - Types of classifiers
- Evaluation of Classifiers
  - Evaluation criteria
  - Model evaluation
    - Cross-Validation
    - Confusion matrices
    - Accuracy, Precision, Recall, and F1-score, the ROC Curve
- Classification algorithms: Naïve Bayes
- **Other types of classification**



Multiclass Classification

- Binary classifiers distinguish between two classes
- Multiclass classifiers (a.k.a. multinomial classifiers) can distinguish between more than two classes
- Some algorithms are capable of handling multiple classes directly
  - Random Forest classifiers or naive Bayes classifiers
- Others are strictly binary classifiers
  - Support Vector Machine classifiers or Linear classifiers
- There are various strategies to perform multiclass classification using multiple binary classifiers

Multiclass Classification: OvA strategy

- One way to create a system that can classify the digit images into 10 classes (from 0 to 9) is to train 10 binary classifiers one for each digit
  - a 0-detector, a 1-detector, a 2-detector, and so on
- Then when you want to classify an image, you get the decision score from each classifier for that image and you select the class whose classifier outputs the highest score
- This is called the one-versus-all (OvA) strategy
  - also called one-versus-the-rest

Multiclass Classification: OvO strategy

- Another strategy is to train a binary classifier for every pair of digits
  - one to distinguish 0s and 1s, another to distinguish 0s and 2s, another for 1s and 2s, and so on
- This is called the one-versus-one (OvO) strategy
  - If there are N classes, you need to train $N \times (N - 1) / 2$ classifiers
- The main advantage of OvO is that each classifier only needs to be trained on the part of the training set for the two classes that it must distinguish

Multiclass Classification: What strategy?

- Some algorithms scale poorly with the size of the training set
  - such as Support Vector Machine classifiers
- For these algorithms OvO is preferred
  - it is faster to train many classifiers on small training sets than training few classifiers on large training sets
- For most binary classification algorithms OvA is preferred

Multilabel Classification (1)

- Until now each instance has always been assigned to just one class
- In some cases you may want your classifier to output multiple classes for each instance
- For example, consider a face-recognition classifier
  - what should it do if it recognizes several people on the same picture?
  - It should attach one label per person it recognizes

Multilabel Classification (2)

- A face-recognition classifier
- The classifier has been trained to recognize three faces
  - Alice, Bob, and Charlie
- When it is shown a picture of Alice and Charlie, it should output [1, 0, 1]
  - meaning “Alice yes, Bob no, Charlie yes”
- Such a classification system that outputs multiple binary labels is called a multilabel classification system

Multilabel Classification: Evaluation (1)

- There are many ways to evaluate a multilabel classifier, and selecting the right metric really depends on your project
- For example, one approach
  - is to measure the F1-score for each individual label
    - or any other binary classifier
  - then simply compute the average score
- This assumes that all labels are equally important, which may not be the case

Multilabel Classification: Evaluation (2)

- If you have many more pictures of Alice than of Bob or Charlie
- you may want to give more weight to the classifier’s score on pictures of Alice
- One simple option is to give each label a weight equal to its support
  - the number of instances with that target label

Multioutput Classification

A multioutput-multiclass classification (or simply multioutput classification) is simply a generalization of multilabel classification where each label can be multiclass

- Let's build a system that removes noise from images
  - It will take as input a noisy digit image, and it will (hopefully) output a clean digit image, represented as an array of pixel intensities
- Notice that the classifier's output is multilabel and each label can have multiple values
  - Multilabel: one label per pixel
  - Multiple values: pixel intensity ranges from 0 to 255