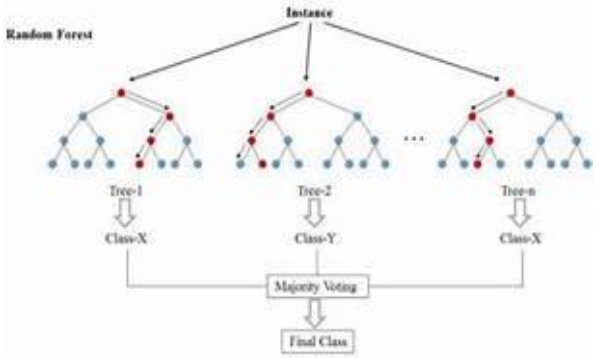
Text Classification (II)

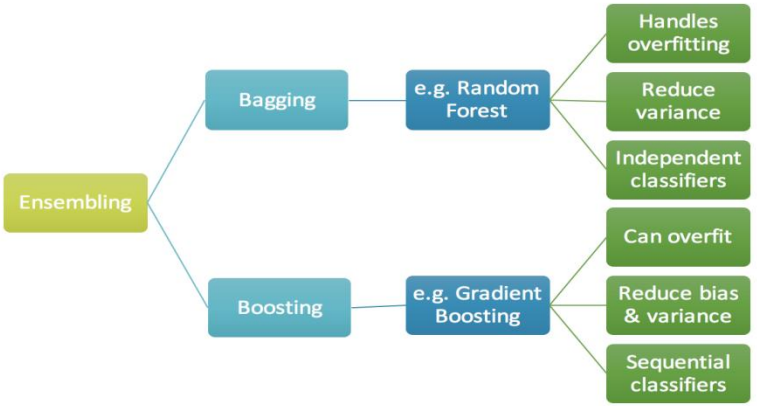
Text and Web Mining (H6751)

School of Communication and Information

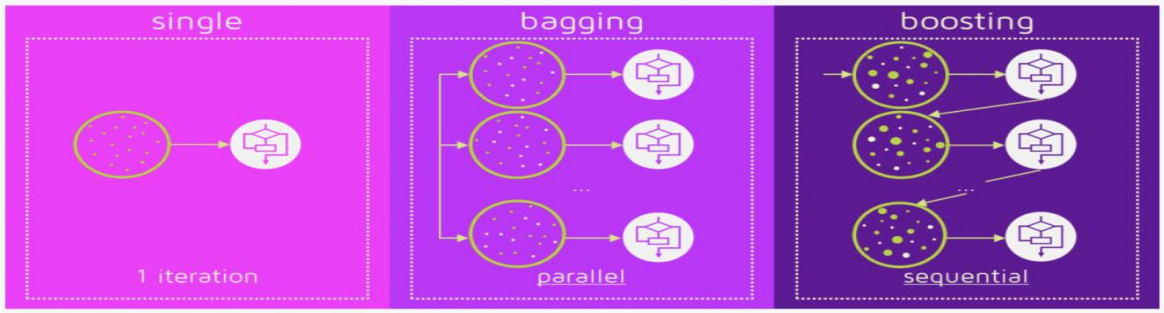
Bagging and Boosting

- **Bagging** is an ensemble technique in which we build many *independent* predictors/models/learners and combine them using some model averaging techniques. (e.g. weighted average, majority vote or normal average)





- **Boosting** is an ensemble technique in which the predictors are not made independently, but sequentially. This technique employs the logic in which the subsequent predictors learn from the mistakes of the previous predictors.



Boosting

- In Boosting, the predictors can be chosen from a range of models like decision trees, regressors, classifiers etc. Because new predictors are learning from mistakes committed by previous predictors, it takes less time/iterations to reach close to actual predictions.
- Choice of stopping criteria is important or it could lead to overfitting on training data. **Gradient Descent** is an example of boosting algorithm.
- Mean squared error (MSE) as loss is defined as:

$$Loss = MSE = \sum (y_i - y_i^p)^2$$

where, y_i = ith target value, y_i^p = ith prediction

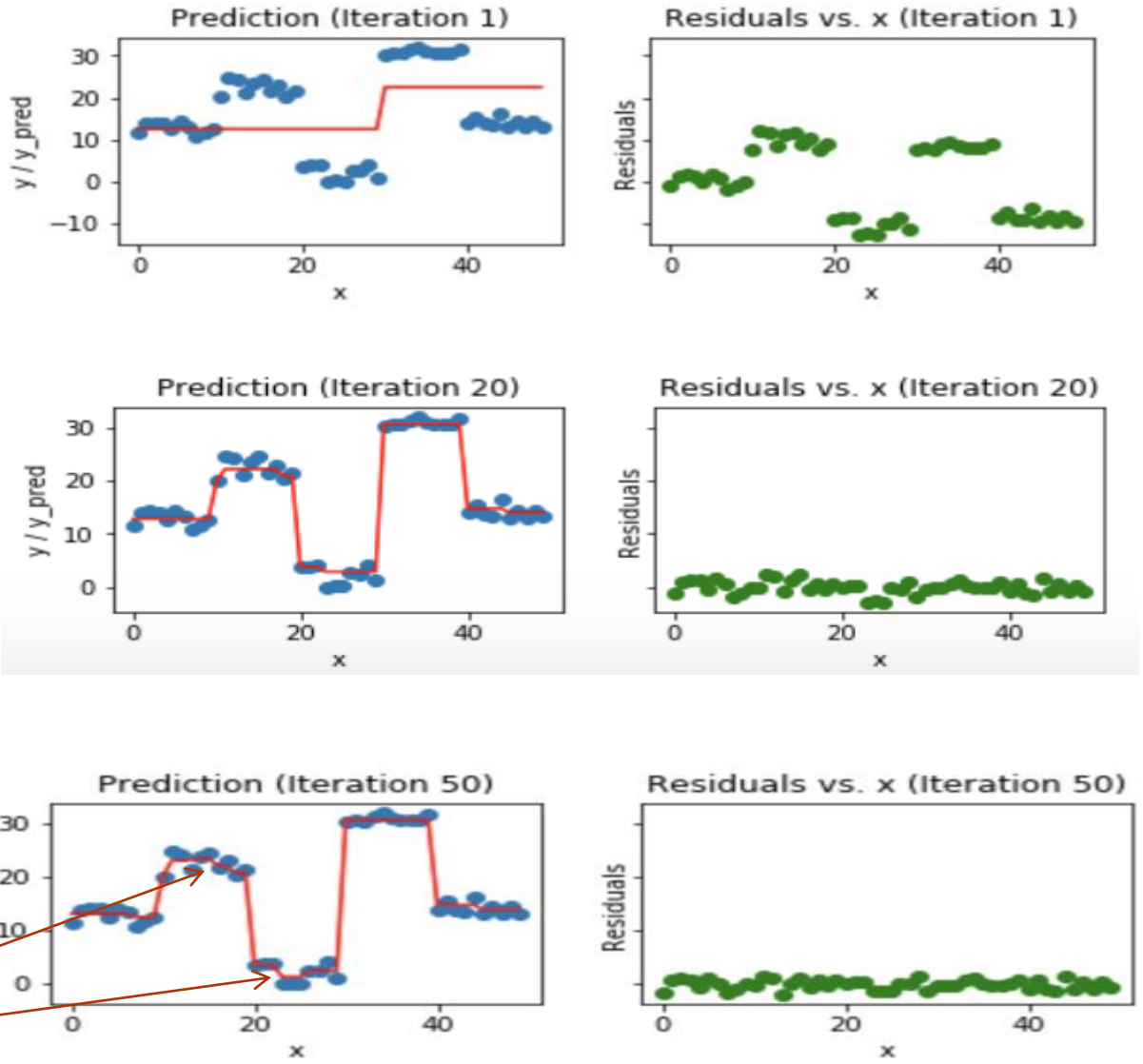
- Gradient Descent Algorithm:

$$\underset{\text{new weight}}{\boldsymbol{W^{(k+1)}}} = \underset{\text{old weight}}{\boldsymbol{W^{(k)}}} - \boldsymbol{\alpha}(\boldsymbol{Loss})$$

$\boldsymbol{\alpha}$: learning rate

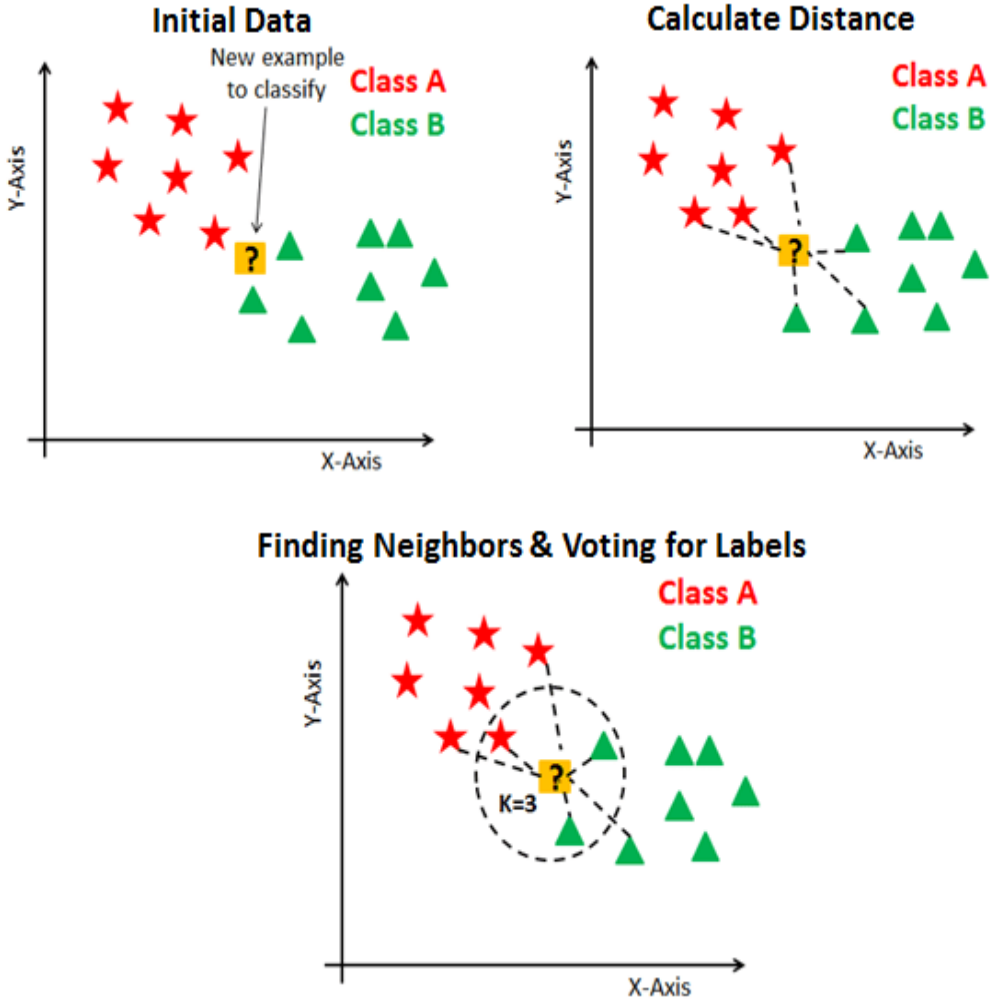
Boosting

Blue dots (left) plots are input (x) vs. output (y) • Red line (left) shows values predicted by decision tree • Green dots (right) shows residuals vs. input (x) for ith iteration • Iteration represent sequential order of fitting gradient boosting tree



Overfitting occurs at high iteration

Similarity and Nearest-Neighbor Methods: K-Nearest Neighbor (K-NN)



- Requires:
  - Set of stored "training" records
  - Distance metric to compute distance between records
  - K = odd number of nearest neighbors
- To classify an unknown record:
  - Compute distance to other training records
  - Identify k nearest neighbors
  - Use class labels of nearest neighbors to determine the class label of unknown record (e.g., by taking majority vote)

Document Similarity

- The most elementary measure of **similarity** is to **count the number of words that two documents have in common**.
- We can then rank the document by similarity, where **the most similar documents have the most shared words**.
- Each document can be represented as **a vector of numbers**. That vector is compared to all the other vectors, and **a score for similarity** is computed.
- The nearest-neighbor method requires **no special effort to learn from the data** (i.e. no supervised learning process).
  - E.g., Naïve Bayes model needs to estimate conditional probabilities of words.

Document Similarity

- There are several ways of representing the measurements in the spreadsheets, including **binary**, **frequency**, and **tf-idf**.
  - E.g., $X \cdot Y = <1, 0, 1, 1> \cdot <1, 0, 1, 0> = 2$ ← 1*1 + 0*0 + 1*1 + 1*0

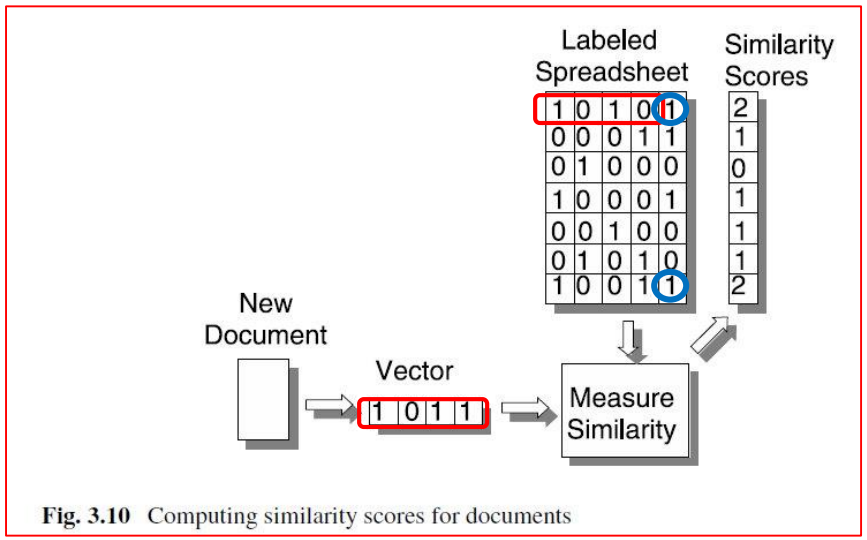


Fig. 3.10 Computing similarity scores for documents

When K=2, output label becomes class 1.

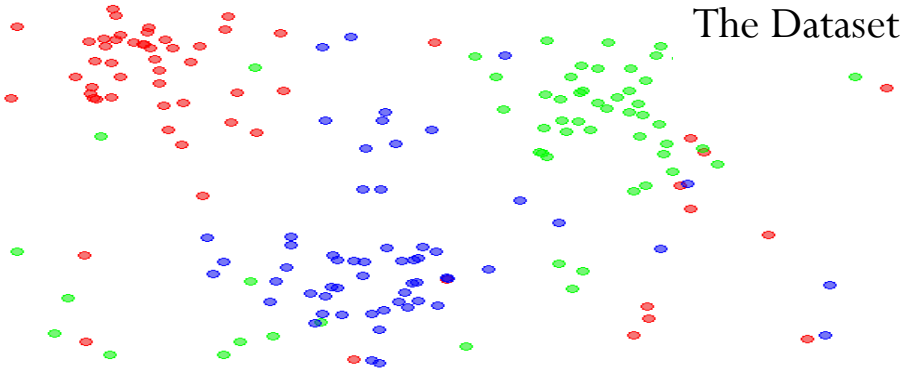
The similarity score of a new French document to "fra" will be the highest.

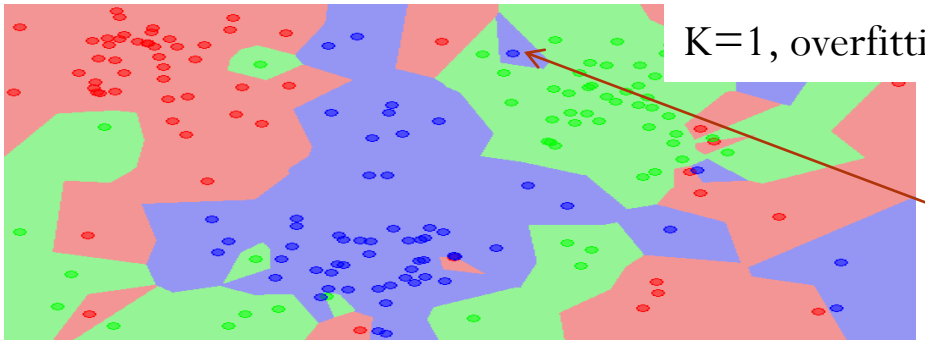
- E.g., Use in Language Identification: k = 1

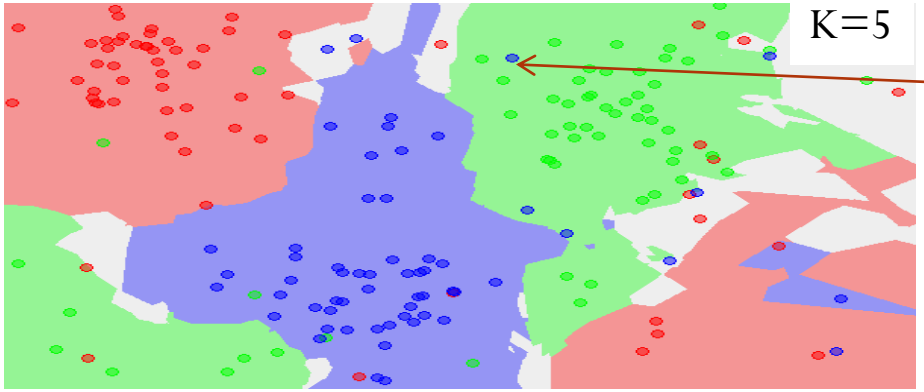
A new Document in French

language	text
deu	Es bleibt wohl das Geheimnis von Prof. Hack, warum er sich dazu hinreißen ließ, mit seinem Leserbrief eine unsinnige Diskussion z
eng	Intrigued I started to work it out. But she couldn't sleep, knowing that her mother was sitting alone somewhere in a big foreign airport t
fra	Le président de l'OM, Jean-Claude Dassier, y confirme à son homologue toulousain, Olivier Sadran, "l'intérêt de l'Olympique de Marse
por	Você deve chamar o vendedor e dizer que ele não está cumprindo as metas. Você está estabelecendo uma criteriosa base de valore
spa	Doctoras, enfermeras, bailarinas!; cantantes, maestras, mams, hasta domadoras de leones pero, habr alguien que de nia haya pen:

K-NN

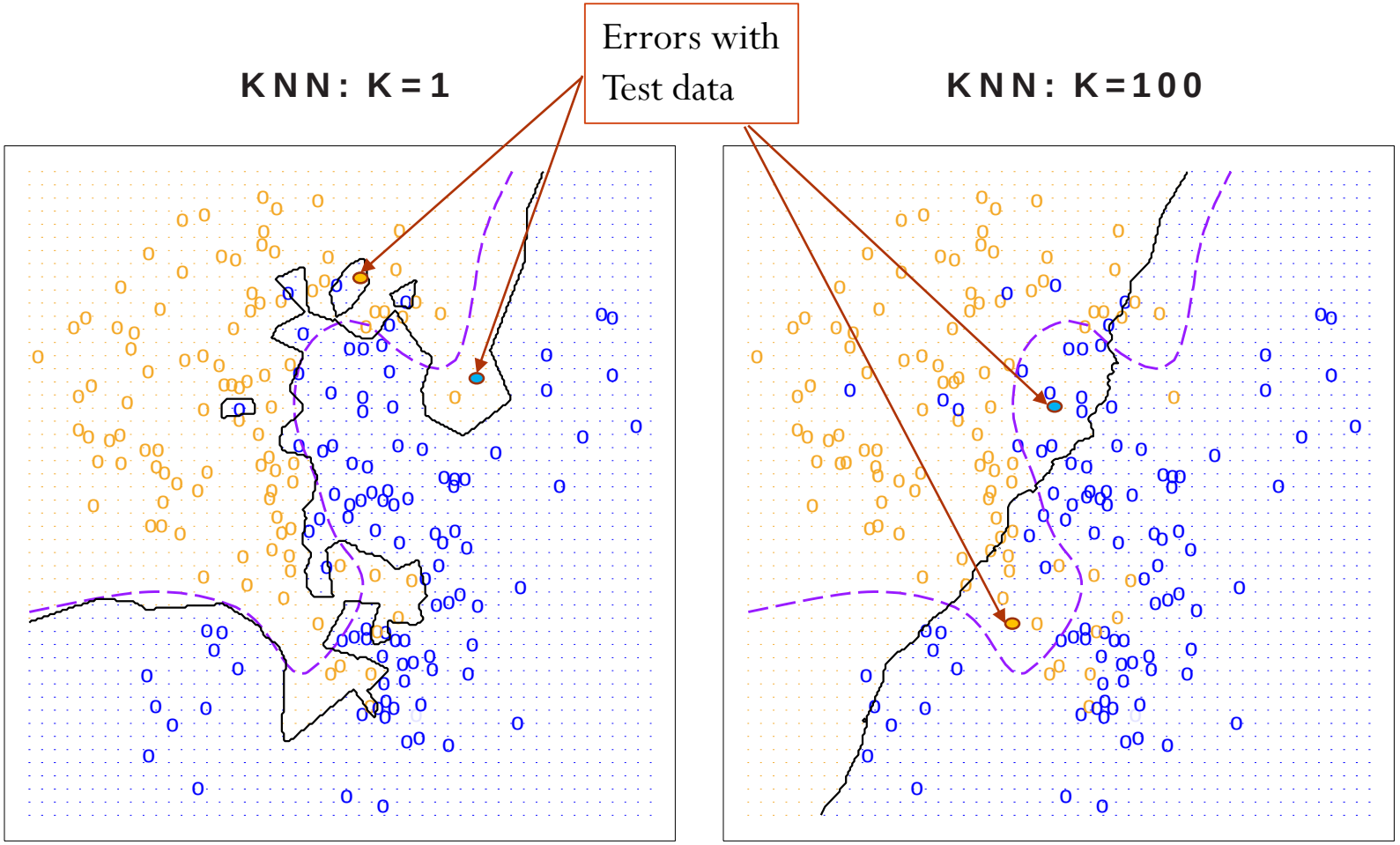






- Choice of K
  - Small K => overfitting
  - Large K = under fitting, may include points from other classes
- Simple to implement but slow and expensive

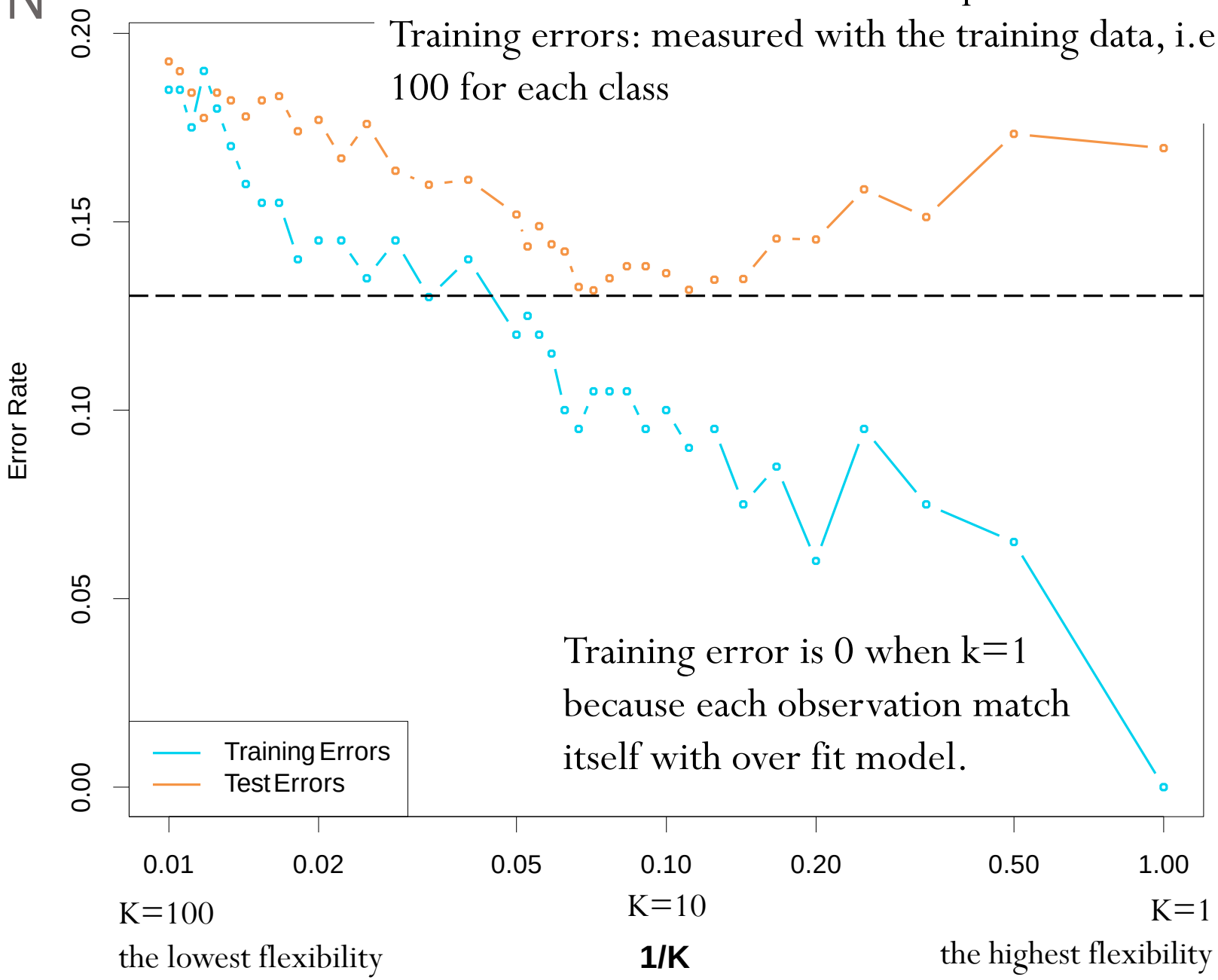
Example: K-Nearest Neighbors in two dimensions



When K=1, too wiggly (over fit)

When K=100, too rough (under fit to 2 partitions => in high dimensional data).

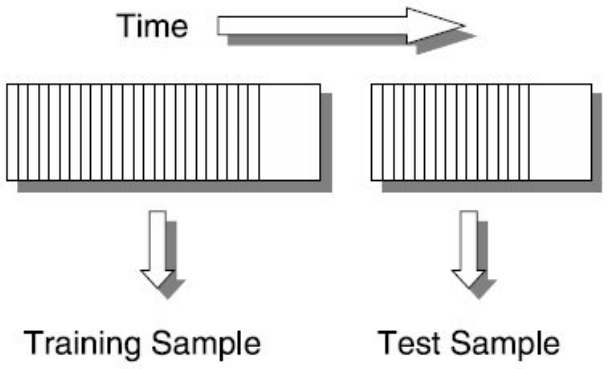
K-NN


Test errors: measured with a new separate test data
Training errors: measured with the training data, i.e. 100 for each class
Error Rate
0.00
0.05
0.10
0.15
0.20
Training Errors
Test Errors
Training error is 0 when k=1 because each observation match itself with over fit model.
0.01
0.02
0.05
0.10
0.20
0.50
1.00
K=100
the lowest flexibility
K=10
1/K
K=1
the highest flexibility

Evaluation of Performance

- To evaluate the performance of a solution, train on one sample and test on another sample.
  - Typically, data might be **randomly divided** into two parts: **one for training** and **one for testing**.
- Some applications of evaluation of text-mining solutions **orders a sample by time** and uses the earlier documents for training and the later documents for testing. (E.g., Stock Prediction)
- This is illustrated in Fig. 3.23, where the sample is divided by time, not randomly.

Fig. 3.23 Partitioning documents into training and test sets



Evaluation of Performance

- Because document collections are typically large, **high precision** is often **more valued** (e.g., Search engine supports high precision).
  - For high precision, the computer's positive decisions are usually correct, but it may fail to catch all positives (this is measured by recall).
- Thus, if a program identifies **spam e-mail with high precision and low recall**, it may often leave spam in your Inbox (low recall), but when it puts a spam document in the trash, it is usually correct (high precision).
  - Obviously, it is more dangerous to move a good message into the trash than not to detect some spam e-mail.
  - So precision takes priority over recall.

Evaluation of Performance

- Is it possible to adjust the precision and recall of a classifier?
- Since precision and recall measure different kinds of errors, **if the overall error rate remains the same, increasing the precision** (reducing one kind of error) **lowers the recall** (increases the other kind of error).
- Most classifiers have a way of making a ***precision–recall trade-off*** by a simple variation of a constant.
- For example, the process is as follows:
  - For linear models (e.g., SVM), the threshold can be changed from zero to a different value. Lower values would help recall, and higher values would boost precision (E.g., Spam detection).

More Than Two Classes: Sets of binary classifiers

- Dealing with **any-of** or **multivalue** classification
  - A document can belong to **0, 1, or >1 classes**.
  - E.g., sports, education, and travel.
- For each class $c \in C$
  - Build a classifier γ_c to distinguish c from all other classes $c' \in C$ (E.g., sports vs. others)
- Given test doc d,
  - Evaluate it for membership in each class using each γ_c
  - d belongs to **any** class for which γ_c returns true.

More Than Two Classes: Sets of binary classifiers

- **One-of** or **multinomial** classification
  - Classes are mutually exclusive: each document in **exactly one class**
  - E.g., positive, neutral, and negative classes
- For each class $c \in C$
  - Build a classifier γ_c to distinguish c from all other classes $c' \in C$
- Given test doc d,
  - Evaluate it for membership in each class using each γ_c
  - d belongs to the **one** class with **maximum score**.

Confusion Matrix

- For each pair of classes $<c_1,c_2>$, how many documents from c_1 were incorrectly assigned to c_2?
  - $c_{3,2}$: 90 wheat documents incorrectly assigned to poultry

Docs in test set	Assigned UK	Assigned poultry	Assigned wheat	Assigned coffee	Assigned interest	Assigned trade
True UK	95	1	13	0	1	0
True poultry	0	1	0	0	0	0
True wheat	10	90	0	1	0	0
True coffee	0	0	0	34	3	7
True interest	0	1	2	13	26	5
True trade	0	0	2	14	5	10

Per class evaluation measures

Docs in test set	Assigned UK	Assigned poultry	Assigned wheat	Assigned coffee	Assigned interest	Assigned trade
True UK	95	1	13	0	1	0
True poultry	0	1	0	0	0	0
True wheat	10	90	0	1	0	0
True coffee	0	0	0	34	3	7
True interest	0	1	2	13	26	5
True trade	0	0	2	14	5	10

Recall:

Fraction of docs in class *i* classified correctly:

- E.g., For UK, => 95 / (95 + 1 + 13 + 0 + 1 + 0)

$$\frac{c_{ii}}{\sum_j c_{ij}}$$

Precision:

Fraction of docs assigned class *i* that are actually about class *i*:

- E.g., For UK, => 95 / (95 + 0 + 10 + 0 + 0 + 0)

$$\frac{c_{ii}}{\sum_j c_{ji}}$$

Accuracy: (1 - error rate)

Fraction of docs classified correctly:

- E.g., For all classes => (95 + 1 + 0 + 34 + 26 + 10) / (95 + 0 + 10 + 0 + 0 + 0) .... (0 + 0 + 0 + 7 + + 10)

$$\frac{\sum_i c_{ii}}{\sum_j \sum_i c_{ij}}$$

Micro- vs. Macro-Averaging

- If we have more than one class, how do we combine multiple performance measures into **one quantity**?
- **Macroaveraging**: Compute performance for each class, then average.
- **Microaveraging**: Collect decisions for all classes, compute contingency table, evaluate.

Micro- vs. Macro-Averaging: Example

- If we have more than one class, how do we combine multiple performance measures into **one quantity**?
- **Macroaveraging**: Compute performance for each class, then average.
- **Microaveraging**: Collect decisions for all classes, compute contingency table, evaluate.

Class 1

	Truth: yes	Truth: no
Classifier: yes (selected)	10	10
Classifier: no (not selected)	10	970

Class 2

	Truth : yes	Truth: no
Classifier: yes	90	10
Classifier: no	10	890

Micro Ave. Table

	Truth: yes	Truth: no
Classifier : yes	100	20
Classifier : no	20	1860

10+90 970+890

- Macroaveraged precision:

  (0.5 {<=**10/20**} + 0.9 {<=**90/100**}) / 2 = 0.7

- Microaveraged precision: 100/120 = .83
- Microaveraged score is dominated by score on common classes

Possible Test Scenario

- Possible test scenario (1):
  - **Train Set, Validation Set**: build a model with Train Set, and use validation set for parameter tuning, and select good parameters
  - **Test set**: build a model using Train Set + Validation Set, and apply it to Test set.
- Possible test scenario (2):
  - **Train Set**: use cross-validation for parameter tuning, and select good parameters
  - **Test set**: build a model using Train Set, and apply it to Test set.
- Possible test scenario (3):
  - **Development Set**: perform parameter tuning, and select good parameters
  - **Dataset (Training + Test set)**: conduct cross validation.

- Depends on availability of data. Separate development, train, validation, and test set support generalization of model.

Building a text classifier for real applications: Very little data?

- Use Naïve Bayes
  - Naïve Bayes is a "high-bias" algorithm – no matter how you set the parameters, you can't get a good training set error.
  - Simple model function => don't need many training data.
- Get more labeled data
  - Find clever ways to get humans to label data for you
- Try semi-supervised training methods:
  - The algorithm first trains a classifier using the available labeled documents, and labels the unlabeled documents with the learned classifier (need to verify the newly labelled documents).
  - It then trains a new classier using the labels for all the documents.

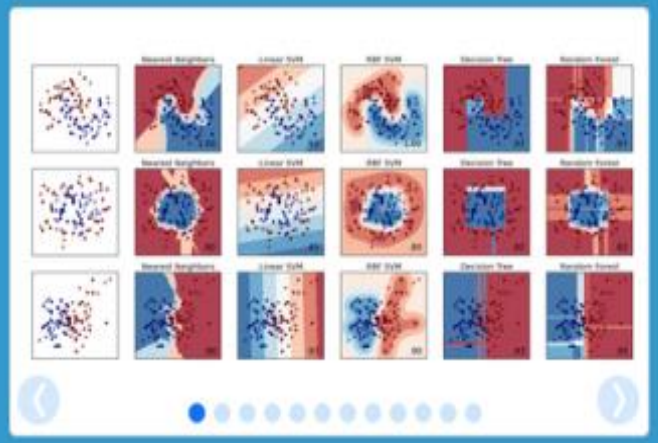
A reasonable to huge amount of data?

- Can achieve high accuracy!
- Decision Trees – user-interpretable rules.
- At a cost:
  - SVMs (train time), random forest (train time), or k-NN (test time) can be too slow.
  - Regularized logistic regression can be somewhat better.
- With enough data
  - Classifier may not matter.

Scikit -learn

Scikit-learn is a free software machine learning library for the Python programming language

- Released in 2007, scikit-learn is one of the most popular open source machine learning libraries for Python. Built on **NumPy**, **SciPy**, and **matplotlib**.
- Scikit-learn provides algorithms for machine learning tasks including classification, regression, dimensionality reduction, and clustering. It also provides modules for extracting features, processing data, and evaluating models.
- Installation >> conda install -c anaconda scikit-learn=0.21.2



scikit-learn

Machine Learning in Python

- Simple and efficient tools for data mining and data analysis
- Accessible to everybody, and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license

Classification

Identifying to which set of categories a new observation belong to.

Applications: Spam detection, Image recognition.
Algorithms: *SVM*, *nearest neighbors*, *random forest*, ... — *Examples*

Regression

Predicting a continuous value for a new example.

Applications: Drug response, Stock prices.
Algorithms: *SVR*, *ridge regression*, *Lasso*, ... — *Examples*

Clustering

Automatic grouping of similar objects into sets.

Applications: Customer segmentation, Grouping experiment outcomes
Algorithms: *k-Means*, *spectral clustering*, *mean-shift*, ... — *Examples*

Dimensionality reduction

Reducing the number of random variables to consider.

Applications: Visualization, Increased efficiency
Algorithms: *PCA*, *Isomap*, *non-negative matrix factorization*. — *Examples*

Model selection

Comparing, validating and choosing parameters and models.

Goal: Improved accuracy via parameter tuning
Modules: *grid search*, *cross validation*, *metrics*. — *Examples*

Preprocessing

Feature extraction and normalization.

Application: Transforming input data such as text for use with machine learning algorithms.
Modules: *preprocessing*, *feature extraction*. — *Examples*

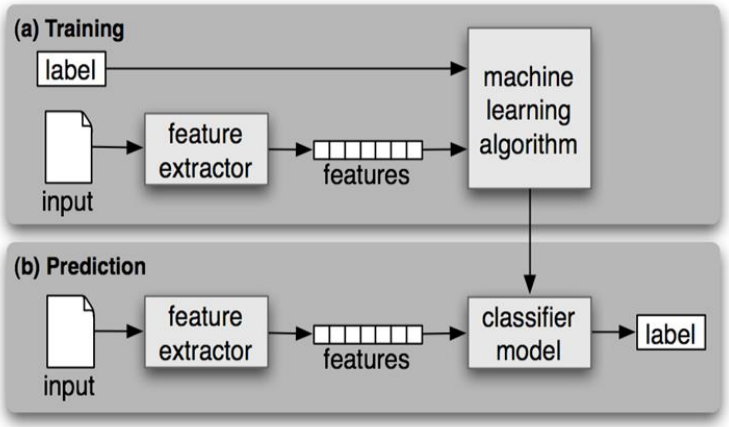
Pandas

- The **SciPy** library is focused on **modeling** data. It is not focused on loading, manipulating and summarizing data. For these, **NumPy** and **Pandas** are used.

- Pandas is an open source library that provides data structures and analysis tools for Python - used for **importing data** and calculating summary **statistics**.

  - Anaconda has Pandas pre-installed.
  - Alternative installation (Numpy has to be installed first) – pip install pandas

5 generic steps to classification



1. **Corpus** collection.
2. Create **features vectors** –
   - E.g., using **TF-IDF** – CountVectorizer, TfidfTransformer, TfidfVectorizer
3. **Training** => Fit and transform the dataset (vectors).
   1. **.fit()** method performs the **training** and requires the training data processed by the vectorizer and correct class labels.
   2. **.transform()** method will **create the vocabulary** (i.e. the list of words/features) and the **feature weights** from the training data.
   3. A **.transform()** on the test data will create the feature weights for the **test data**, using the same vocabulary as the training data.
4. Perform **prediction (classification), cross validation**
5. Evaluate **results** –precision, recall, F1, grid search.

Step 1 - Corpus Representation

- A collection of documents is called a **corpus**. E.g. the following two documents:

corpus = ['UNC played Duke in basketball', **'Duke** lost the **basketball** game']

- Corpus's unique words comprise its **vocabulary (**Eight unique words). The bag-of-words model uses a **feature vector** with an element for each of the words in the corpus's vocabulary to represent each document (ignores word order and grammar).

	basketball	duke	game	in	lost	played	the	unc
D1	1	1	0	1	0	1	0	1
D2	1	1	1	0	1	0	1	0

The **number of elements** (in this case 8) that comprise a feature vector is called the vector's **dimension**.

Step 1 - Corpus Collection

- Using **pandas** to load corpus

```
>>> import numpy as np
>>> import pandas as pd
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> from sklearn.linear_model.logistic import LogisticRegression
>>> from sklearn.cross_validation import train_test_split, cross_val_
score
```

Corpus loading

```
>>> df = pd.read_csv('data/sms.csv')
>>> X_train_raw, X_test_raw, y_train, y_test = train_test_
split(df['message'], df['label'])
```

- Assigning training and test data using **train_test_split()**

SPAM Classification Data Sample

label	message
0	Go until jurong point, crazy.. Available only in bugis n great world la e buffet... Cine there got amore wat...
0	Ok lar... Joking wif u oni...
1	Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005. Text FA to 87121 to receive entry question(std txt rate)T&C's apply 08452810075over18's
0	U dun say so early hor... U c already then say...
0	Nah I don't think he goes to usf, he lives around here though
1	FreeMsg Hey there darling it's been 3 week's now and no word back! I'd like some fun you up for it still? Tb ok! XxX std chgs to send, Â£1.50 to rcv
0	Even my brother is not like to speak with me. They treat me like aids patent.
0	As per your request 'Melle Melle (Oru Minnaminunginte Nurungu Vettam)' has been set as your callertune for all Callers. Press *9 to copy your friends Callertune
1	WINNER!! As a valued network customer you have been selected to receivea Â£900 prize reward! To claim call 09061701461. Claim code KL341. Valid 12 hours only.
1	Had your mobile 11 months or more? U R entitled to Update to the latest colour mobiles with camera for Free! Call The Mobile Update Co FREE on 08002986030
0	I'm gonna be home soon and i don't want to talk about this stuff anymore tonight, k? I've cried enough today.
1	SIX chances to win CASH! From 100 to 20,000 pounds txt> CSH11 and send to 87575. Cost 150p/day, 6days, 16+ TsandCs apply Reply HL 4 info
1	URGENT! You have won a 1 week FREE membership in our Â£100,000 Prize Jackpot! Txt the word: CLAIM to No: 81010 T&C www.dbuk.net LCCLTD POBOX 4403LDNW1A7RW18
0	I've been searching for the right words to thank you for this breather. I promise i wont take your help for granted and will fulfil my promise. You have been wonderful and a b
0	I HAVE A DATE ON SUNDAY WITH WILL!!

Step 2 – Features Extraction

- The documents in the corpus can be represented by **feature vectors**:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> corpus = [
>>> 'UNC played Duke in basketball',
>>> 'Duke lost the basketball game'
>>> ]
```

Module under Preprocessing

	basketball	duke	game	in	lost	played	the	unc
D1	1	1	0	1	0	1	0	1
D2	1	1	1	0	1	0	1	0

```
>>> vectorizer = CountVectorizer()
>>> print vectorizer.fit_transform(corpus).todense()
>>> print vectorizer.vocabulary_
```

transform => create the **vocabulary** and **feature weights** (word count)

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

todense() format

```
{u'duke': 1, u'basketball': 0, u'lost': 4, u'played': 5, u'game': 2, u'unc': 7,
u'in': 3, u'the': 6}
```

vocabulary

Step 2 - Stop-word filtering

- The **CountVectorizer** class can filter stop words using the **stop_words** keyword argument.
- The feature vectors have now **fewer dimensions**, and the first two document vectors are **even** more similar to each other.

```
> from sklearn.feature_extraction.text import CountVectorizer
> corpus = [
> 'UNC played Duke in basketball',
> 'Duke lost the basketball game']
> vectorizer = CountVectorizer(stop_words='english')
> print vectorizer.fit_transform(corpus).todense()
> print vectorizer.vocabulary_
[[1 1 0 0 1 1]
[1 1 1 1 0 0 ]]
```

6 dimensions vs 8 ('in' and 'the' removed)

{**u'duke'**: 1, **u'basketball'**: 0, u'lost': 3, u'played': 4, u'game': 2, u'unc': 5}

Step 2 - TF and TF-IDF

- **Term frequency (TF)** – raw frequency of a term in document => $tf(t,d)$
- Words might appear with the same frequency in two documents, but the documents could be dissimilar if one is **many times larger** than the other.
  - "I love **apple**" vs "I love fruits especially **apples**, **oranges**, **pears**, and …"
- Mitigate problem by using **normalized** term frequency weights => $tf(t,d) = \frac{f(t,d)}{||x||}$
- $f(t, d)$ is the frequency of term in document d and x is the L2 norm of the count vector of terms in the document => $\sqrt{x_1^2 + x_2^2 + x_3^2 + \ \ldots}$
- **TF-IDF** is used instead of the raw frequencies of a term to scale down impact of frequently occurring terms in a given **corpus**, which are empirically less informative than features that occur in a small fraction of the training corpus.
- **Inverse Document Frequency (IDF)** measures how rare or common a word is in a corpus => more rare more significant.

$$(\boldsymbol{i})\ \ idf(d,t) = \log\left[\frac{(1+n)}{(1+df(d,t))}\right] + 1$$

$$\textit{TF-IDF} = tf(t,d).idf(d,t) = \frac{f(t,d)}{||x||} . \log\left[\frac{(1+n)}{(1+df(d,t))}\right] + 1$$

Terms that occur in no documents (df=0) will not lead to division by zero.

terms that occur in all documents (df=n) leading to log(1)=0, will not be entirely ignored.

The resulting tf-idf vectors are then normalized based on the tf-idf score of each term.

Step 2 - CountVectorizer and TfidfTransformer

- CountVectorizer - creation of feature vectors that encode the frequencies of words (term frequencies)

```
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer
corpus = [
    'The dog ate a sandwich and I ate a sandwich',
    'The wizard transfigured a sandwich'
]
vectorizer = CountVectorizer(stop_words='english')
transformer = TfidfTransformer(use_idf=False)
transformerIDF = TfidfTransformer(use_idf=True)
X = vectorizer.fit_transform(corpus)
print('Count vectors:\n', X.todense())
print('Vocabulary:\n', vectorizer.vocabulary_)
print('TF vectors:\n', transformer.fit_transform(X).todense())
print('TF-IDF vectors:\n', transformerIDF.fit_transform(X).todense())
```

```
Count vectors:
 [[2 1 2 0 0]
 [0 0 1 1 1]]
Vocabulary:
 {'dog': 1, 'ate': 0, 'sandwich': 2, 'wizard': 4, 'transfigured': 3}
TF vectors:
 [[ 0.66666667  0.33333333  0.66666667  0.          0.        ]
 [ 0.          0.          0.57735027  0.57735027  0.57735027]]
TF-IDF vectors:
 [[ 0.75458397  0.37729199  0.53689271  0.          0.        ]
 [ 0.          0.          0.44943642  0.6316672   0.6316672 ]]
```

- **TfidfTransformer** returns **TF-IDF weight** when the **use_idf** keyword argument is True (default value) (returns **TF weight** if use_idf set to False)

Step 2 - TfidfVectorizer

- Scikit-learn provides a **TfidfVectorizer** class that wraps **CountVectorizer** and **TfidfTransformer**.

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> corpus = [
>>>     'The dog ate a sandwich and I ate a sandwich',
>>>     'The wizard transfigured a sandwich'
>>> ]
>>> vectorizer = TfidfVectorizer(stop_words='english')
>>> print vectorizer.fit_transform(corpus).todense()
[[ 0.75458397  0.37729199  0.53689271  0.          0.        ]
 [ 0.          0.          0.44943642  0.6316672   0.6316672 ]]
```

Computes TF-IDF directly

ate

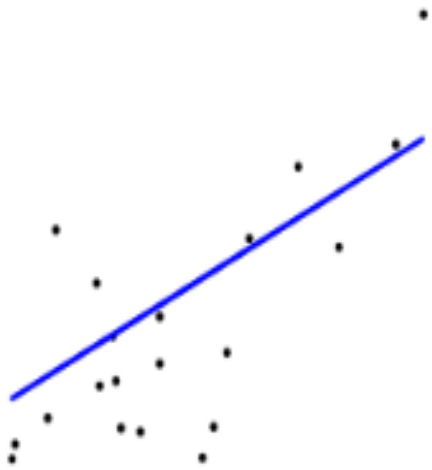
sandwich

- By comparing the TF-IDF weights to the raw term frequencies, we can see that words that are common to many of the documents in the corpus, such as **sandwich**, have been penalized.

Step 3 - Fit and Transform

- The **fit()** method **performs the training** to obtain the model **parameters** and requires the **training data** processed by the vectorizer and correct **class labels.**
- The **transform()** method will create the **vocabulary** (i.e. the list of words/features) and the **feature weights**.
- For example, you fit a linear model to data to get the slope, intercept and line coordinates. Then you use those **parameters** to transform (i.e., map) new or existing values of `x` to `y`.

```
>>> from sklearn import linear_model
>>> regr = linear_model.LinearRegression()
>>> regr.fit(diabetes_X_train, diabetes_y_train)
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1,
 normalize=False)
>>> print(regr.coef_)
[   0.30349955 -237.63931533  510.53060544  327.73698041 -81
4.13170937
  492.81458798  102.84845219  184.60648906  743.51961675   7
6.09517222]
```



Step 3 - fit_transform() method

In the example, the fit() method fits the model to the data showing the parameters of the trained model.

The transform() method transforms the data to a word frequency representation.

```
from sklearn.feature_extraction.text import CountVectorizer
corpus = ['The dog ate a sandwich, the wizard transfigured a sandwich, and I ate a sandwich']
vectorizer = CountVectorizer(stop_words='english')
#print(vectorizer.vocabulary_)

print(vectorizer.fit(corpus))
print(vectorizer.transform(corpus))
print(vectorizer.fit_transform(corpus).todense())
```

The fit_transform() method does both fit and transform steps to the same data.

```
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
        dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
        lowercase=True, max_df=1.0, max_features=None, min_df=1,
        ngram_range=(1, 1), preprocessor=None, stop_words='english',
        strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
        tokenizer=None, vocabulary=None)
```

parameters

```
  (0, 0)        2
  (0, 1)        1
  (0, 2)        3
  (0, 3)        1
  (0, 4)        1
[[2 1 3 1 1]]
```

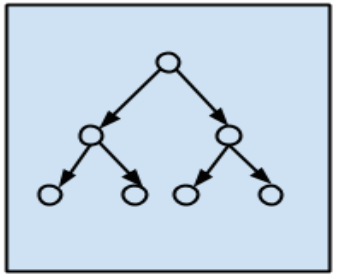
Compressed Sparse Row (CSR) representations

todense() representation

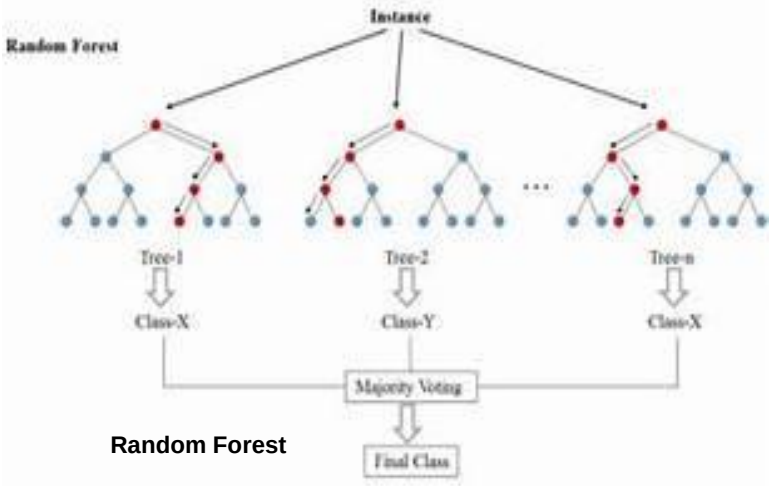
max_df removes terms that appear **too frequently** => "corpus-specific stop words". max_df = 0.50 "ignore terms that appear in **more than 50% of the documents**". Default max_df=1 does not ignore any terms.

min_df removes terms that appear **too infrequently**. min_df = 0.01 means "ignore terms that appear in **less than 1% of the documents**". Default min_df=1 does not ignore any terms.

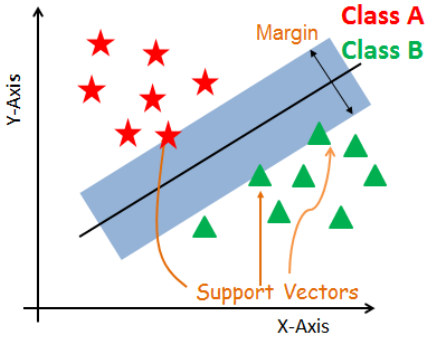
Step 4 – Classification (Models)

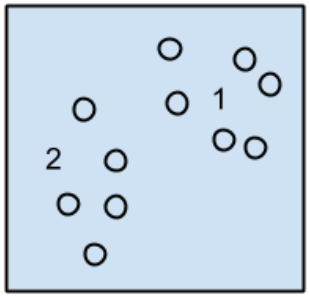


Decision Tree Algorithms

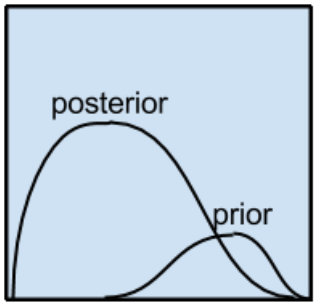


Random Forest

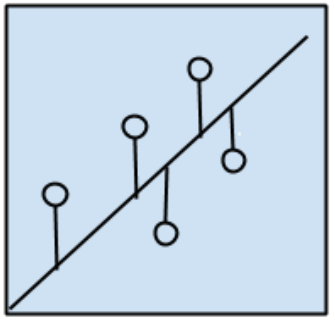




Clustering Algorithms



Bayesian Algorithms



Regression Algorithms

Step 4 - Spam/Ham Classification Example

Use the SMS Spam Classification Data Set from the UCI Machine Learning Repository. The dataset can be downloaded from https://archive.ics.uci.edu/ml/datasets/SMS+Spam+Collection.

Explore dataset and calculate basic summary statistics using pandas

```
>>> import pandas as pd
>>> df = pd.read_csv('data/SMSSpamCollection', delimiter='\t',
header=None)
>>> print df.head()

      0                                                  1
0   ham  Go until jurong point, crazy.. Available only ...
1   ham                      Ok lar... Joking wif u oni...
2  spam  Free entry in 2 a wkly comp to win FA Cup fina...
3   ham  U dun say so early hor... U c already then say...
4   ham  Nah I don't think he goes to usf, he lives aro...
[5 rows x 2 columns]

>>> print 'Number of spam messages:', df[df[0] == 'spam'][0].count()
>>> print 'Number of ham messages:', df[df[0] == 'ham'][0].count()

Number of spam messages: 747
Number of ham messages: 4825
```

Step 4 - Spam/Ham Classification

Import the LogisticRegression Class

```
>>> import numpy as np
>>> import pandas as pd
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> from sklearn.linear_model.logistic import LogisticRegression
>>> from sklearn.cross_validation import train_test_split, cross_val_
score
```

Load the .csv file using pandas and **split the data set into training and test sets**. By default, train_test_split() assigns **75 percent** of the samples to the **training set** and allocates the remaining **25 percent** of the samples to the **test set**:

```
>>> df = pd.read_csv('data/SMSSpamCollection', delimiter='\t',
header=None)
>>> X_train_raw, X_test_raw, y_train, y_test = train_test_split(df[1],
df[0])
```

Assign X=> message, y=> label

Step 4 - Spam/Ham Classification

Create a TfidfVectorizer (CountVectorizer and TfidfTransformer). Fit with the training messages, and transform both training and test messages.

```
>>> vectorizer = TfidfVectorizer()
>>> X_train = vectorizer.fit_transform(X_train_raw)
>>> X_test = vectorizer.transform(X_test_raw)
```

Learn vocabulary and idf, return term-document matrix

(fit and transform to TFIDF vectorizer)

Transform documents to document-term matrix

(why not fit_transform() for X_test?)

Create an instance of LogisticRegression and train the model. The fit() and predict() methods are implemented.

```
>>> classifier = LogisticRegression()
>>> classifier.fit(X_train, y_train)
>>> predictions = classifier.predict(X_test)
>>> for i, prediction in enumerate(predictions[:5]):
>>>     print 'Prediction: %s. Message: %s' % (prediction, X_test_
raw[i])
```

(fitting requires training TF_IDF vectors and labels)

```
Prediction: ham. Message: If you don't respond imma assume you're
still asleep and imma start calling n shit
Prediction: spam. Message: HOT LIVE FANTASIES call now 08707500020
Just 20p per min NTT Ltd, PO Box 1327 Croydon CR9 5WB 0870 is a
national rate call
```

Step 5 – Performance Measurement (Cross validation)

- Scikit learn's **cross_val_score** function will perform the training, testing, and scoring based on the number of folds
- The **average scores** of all the fold scorings will be used as the overall scores of the model.

Precision is 0.992; almost all of the messages that it predicted as spam were actually spam. Recall is lower, indicating that it incorrectly classified approximately 32 percent of the spam messages as ham.

Precision and recall may vary since the training and test data are randomly partitioned.

```
>>> classifier = LogisticRegression()
>>> classifier.fit(X_train, y_train)
>>> precisions = cross_val_score(classifier, X_train, y_train, cv=5,
scoring='precision')
>>> print 'Precision', np.mean(precisions), precisions
>>> recalls = cross_val_score(classifier, X_train, y_train, cv=5,
scoring='recall')
>>> print 'Recalls', np.mean(recalls), recalls

Precision 0.992137651822 [ 0.98717949  0.98666667  1.
0.98684211  1.        ]
Recall 0.677114261885 [ 0.7         0.67272727  0.6         0.68807339
0.72477064]
```

Code Overview

1. Corpus collection
2. Create TF-IDF features vectors
3. Model training Fit and Transform
4. Predict
5. Evaluation

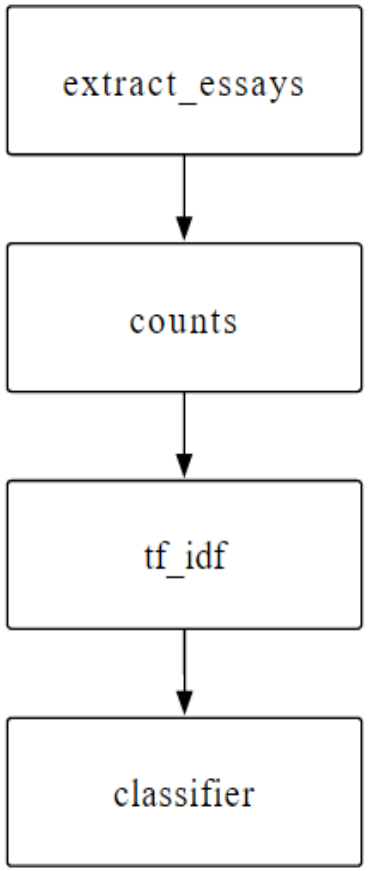
```
>>> import numpy as np
>>> import pandas as pd
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> from sklearn.linear_model.logistic import LogisticRegression
>>> from sklearn.cross_validation import train_test_split, cross_val_
score
>>> df = pd.read_csv('data/sms.csv')
>>> X_train_raw, X_test_raw, y_train, y_test = train_test_
split(df['message'], df['label'])
>>> vectorizer = TfidfVectorizer()
>>> X_train = vectorizer.fit_transform(X_train_raw)
>>> X_test = vectorizer.transform(X_test_raw)
>>> classifier = LogisticRegression()
>>> classifier.fit(X_train, y_train)
>>> precisions = cross_val_score(classifier, X_train, y_train, cv=5,
scoring='precision')
>>> print 'Precision', np.mean(precisions), precisions
>>> recalls = cross_val_score(classifier, X_train, y_train, cv=5,
scoring='recall')
>>> print 'Recalls', np.mean(recalls), recalls

Precision 0.992137651822 [ 0.98717949  0.98666667  1.
0.98684211  1.        ]
Recall 0.677114261885 [ 0.7         0.67272727  0.6         0.68807339
0.72477064]
```

Pipelines

Instead of manually running through steps, and repeating them on the test set, **Pipelines** allow a nice, declarative interface where it's easy to see the **entire model**.

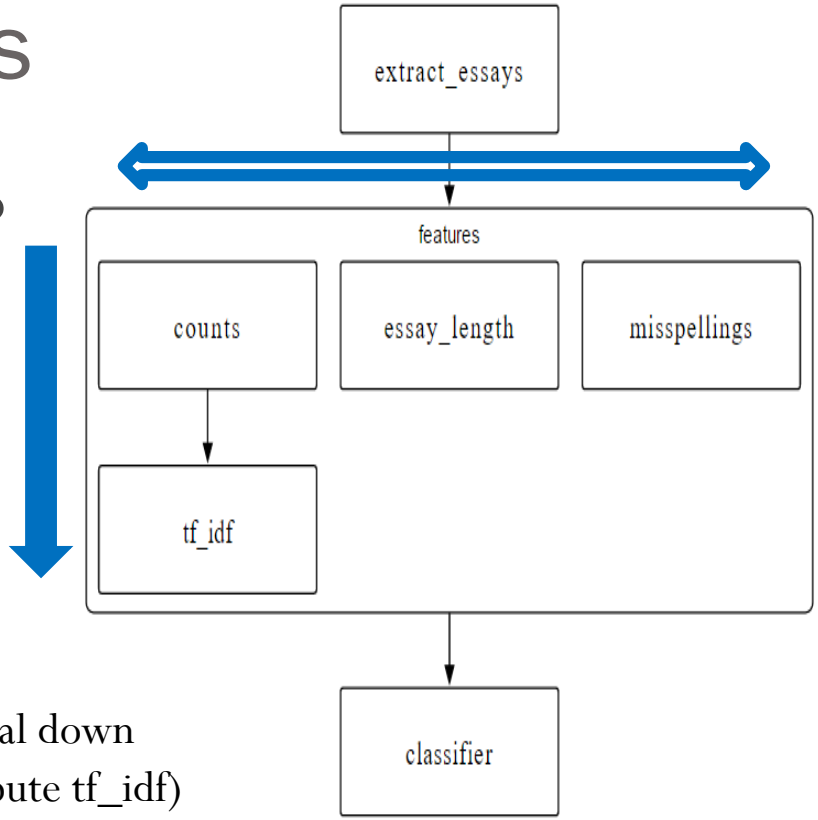
This example extracts the text documents, tokenizes them, counts the tokens, and then performs a tf–idf transformation before passing the resulting features along to a multinomial naive Bayes classifier:

```
pipeline = Pipeline([
  ('extract_essays', EssayExractor()),
  ('counts', CountVectorizer()),
  ('tf_idf', TfidfTransformer()),
  ('classifier', MultinomialNB())
])
```



Pipelines - FeatureUnions

To enable **parallel processes** that need to be performed with the data before putting the results together, use a FeatureUnion




```
pipeline = Pipeline([
  ('extract_essays', EssayExractor()),
  ('features', FeatureUnion([
    ('ngram_tf_idf', Pipeline([
      ('counts', CountVectorizer()),
      ('tf_idf', TfidfTransformer())
    ])),
    ('essay_length', LengthTransformer()),
    ('misspellings', MispellingCountTransformer())
  ])),
  ('classifier', MultinomialNB())
])
```

Horizontal

Vertical down (compute tf_idf)

This example feeds the output of the extract_essays step into each of the **ngram_tf_idf**, **essay_length**, and **misspellings** steps and concatenates their outputs before feeding it into the classifier.

Tuning models with grid search

Hyperparameters are parameters of the model that are not learned (trained). For example, a hyperparameter of the logistic regression SMS classifier is the value of **thresholds** used to remove words that appear too frequently or infrequently.

GridSearchCV() - trains and evaluates a model for **each possible combination** of the hyperparameter values. Disadvantage is that it is computationally costly for even small sets of hyperparameter values – solved through parallel processing.

Tuning models with grid search

Vect => vector
Clf => classifier

Parameters found in respective process in pipeline

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model.logistic import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.model_selection import train_test_split, GridSearchCV

from sklearn.metrics import precision_score, recall_score, accuracy_score

pipeline = Pipeline([
    ('vect', TfidfVectorizer(stop_words='english')),
    ('clf', LogisticRegression())
])
parameters = {
    'vect__max_df': (0.25, 0.5, 0.75),
    'vect__stop_words': ('english', None),
    'vect__max_features': (2500, 5000, 10000, None),
    'vect__ngram_range': ((1, 1), (1, 2)),
    'vect__use_idf': (True, False),
    'vect__norm': ('l1', 'l2'),
    'clf__penalty': ('l1', 'l2'),
    'clf__C': (0.01, 0.1, 1, 10),
}
```

max_df parameter in TfidfVectorizer

Inverse of regularization strength parameter in LogisticRegression

Tuning models with grid search

Use of pipelines

No. of parallel jobs. -1 means use all processors

Grid Search => combination of parameters

Defined in previously

Fit => train the model

Get the best scores / parameters

Predict

Obtain best parameter value

```
if __name__ == "__main__":
    grid_search = GridSearchCV(pipeline, parameters, n_jobs=-1,
verbose=1, scoring='accuracy', cv=3)
    df = pd.read_csv('data/sms.csv')
    X, y, = df['message'], df['label']
    X_train, X_test, y_train, y_test = train_test_split(X, y)
    grid_search.fit(X_train, y_train)
    print 'Best score: %0.3f' % grid_search.best_score_
    print 'Best parameters set:'
    best_parameters = grid_search.best_estimator_.get_params()
    for param_name in sorted(parameters.keys()):
        print '\t%s: %r' % (param_name, best_parameters[param_name])
    predictions = grid_search.predict(X_test)
    print 'Accuracy:', accuracy_score(y_test, predictions)
    print 'Precision:', precision_score(y_test, predictions)
    print 'Recall:', recall_score(y_test, predictions)
```

Tuning models with grid search

Output

```
Fitting 3 folds for each of 1536 candidates, totalling 4608 fits

[Parallel(n_jobs=-1)]: Done  42 tasks      | elapsed:    6.8s
[Parallel(n_jobs=-1)]: Done 192 tasks      | elapsed:   18.9s
[Parallel(n_jobs=-1)]: Done 442 tasks      | elapsed:   39.5s
[Parallel(n_jobs=-1)]: Done 792 tasks      | elapsed:  1.1min
[Parallel(n_jobs=-1)]: Done 1242 tasks      | elapsed:  1.8min
[Parallel(n_jobs=-1)]: Done 1792 tasks      | elapsed:  2.6min
[Parallel(n_jobs=-1)]: Done 2442 tasks      | elapsed:  3.5min
[Parallel(n_jobs=-1)]: Done 3192 tasks      | elapsed:  4.5min
[Parallel(n_jobs=-1)]: Done 4042 tasks      | elapsed:  7.7min
[Parallel(n_jobs=-1)]: Done 4608 out of 4608 | elapsed:  8.5min finished

Best score: 0.983
Best parameters set:
        clf__C: 10
        clf__penalty: 'l2'
        vect__max_df: 0.25
        vect__max_features: 10000
        vect__ngram_range: (1, 2)
        vect__norm: 'l2'
        vect__stop_words: None
        vect__use_idf: True
Accuracy: 0.985652797704
Precision: 0.989847715736
Recall: 0.915492957746
```

Referenced Materials

- NLP, Dan Jurafsky and Christopher Manning, http://www.stanford.edu/~jurafsky/NLPCourseraSlides.html
- Fundamentals of Predictive Text Mining, Sholom M. Weiss, Nitin Indurkhya, and Tong Zhang, Springer.
  - Chapter 3