Text Mining
Project/Lab

Behrang Q. Zadeh
behrangatoffice@gmail.com

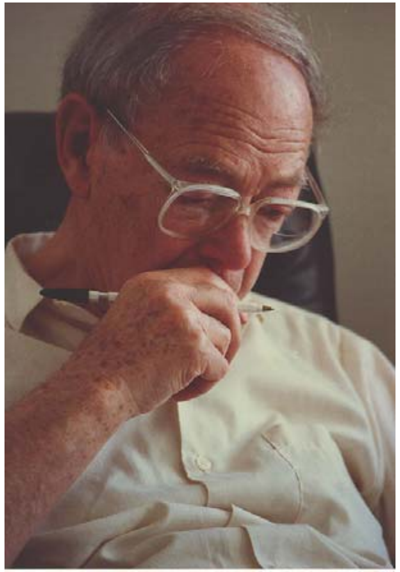
Text Classification

Pattern Recognition and Classification

- Detecting patterns and structures is a central theme in text mining.
- We usually start with the hypothesis that certain observable patterns in text are correlated to a particular task we address in text mining:

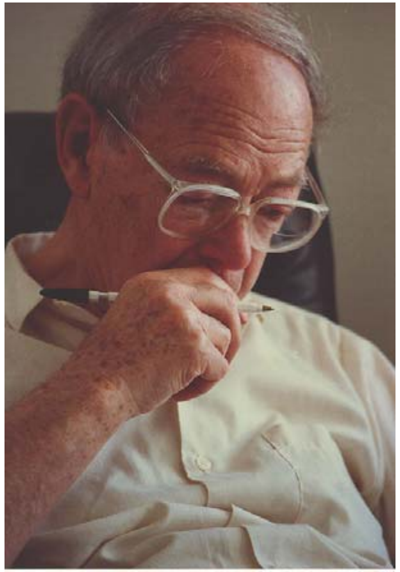
Pattern Recognition and Classification

- Detecting patterns and structures is a central theme in text mining.
- We usually start with the hypothesis that certain observable patterns in text are correlated to a particular task we address in text mining:
  - For example, the **distributional hypothesis** by **Zellig Harris** states that word frequencies are correlated to particular aspects of meaning.



Pattern Recognition and Classification

- Detecting patterns and structures is a central theme in text mining.
- We usually start with the hypothesis that certain observable patterns in text are correlated to a particular task we address in text mining:
  - For example, the **distributional hypothesis** by **Zellig Harris** states that word frequencies are correlated to particular aspects of meaning.
- But, which aspects of form to associate with which aspects of meaning, i.e. where to start?



Goal of this session

- How to identify salient features of text data that are important for a specific task?
- How to construct models of language for an automatic language processing task?
- What can we learn about language from these models?
- What are the examples of machine learning techniques for these tasks?

Classification

- Classification is the task of classifying the elements of a given set into *a number of groups* based on some *criteria:*
  - Text classification is the task of assigning documents to several groups topic labels such as news, sport, etc. (groups: news, sport, etc.)
  - Deciding whether an email is spam or not (groups: spam and not-spam).
  - Deciding whether a given occurrence of the word bank is used to refer to a river bank, or a financial institution (groups: word senses).
- A **classification rule** can be seen as a function that assigns each element of the set to a class/group/label.

Classification

- Binary classification: there are only two groups or class labels.
- Multi-class classification: there are more than two class labels.
- Open-class classification/Clustering: labels are not known in advance.
- Regression: the label variable is a continuous value, e.g. [0-1].
- Sequence classification: a list of inputs are jointly classified.

Classification

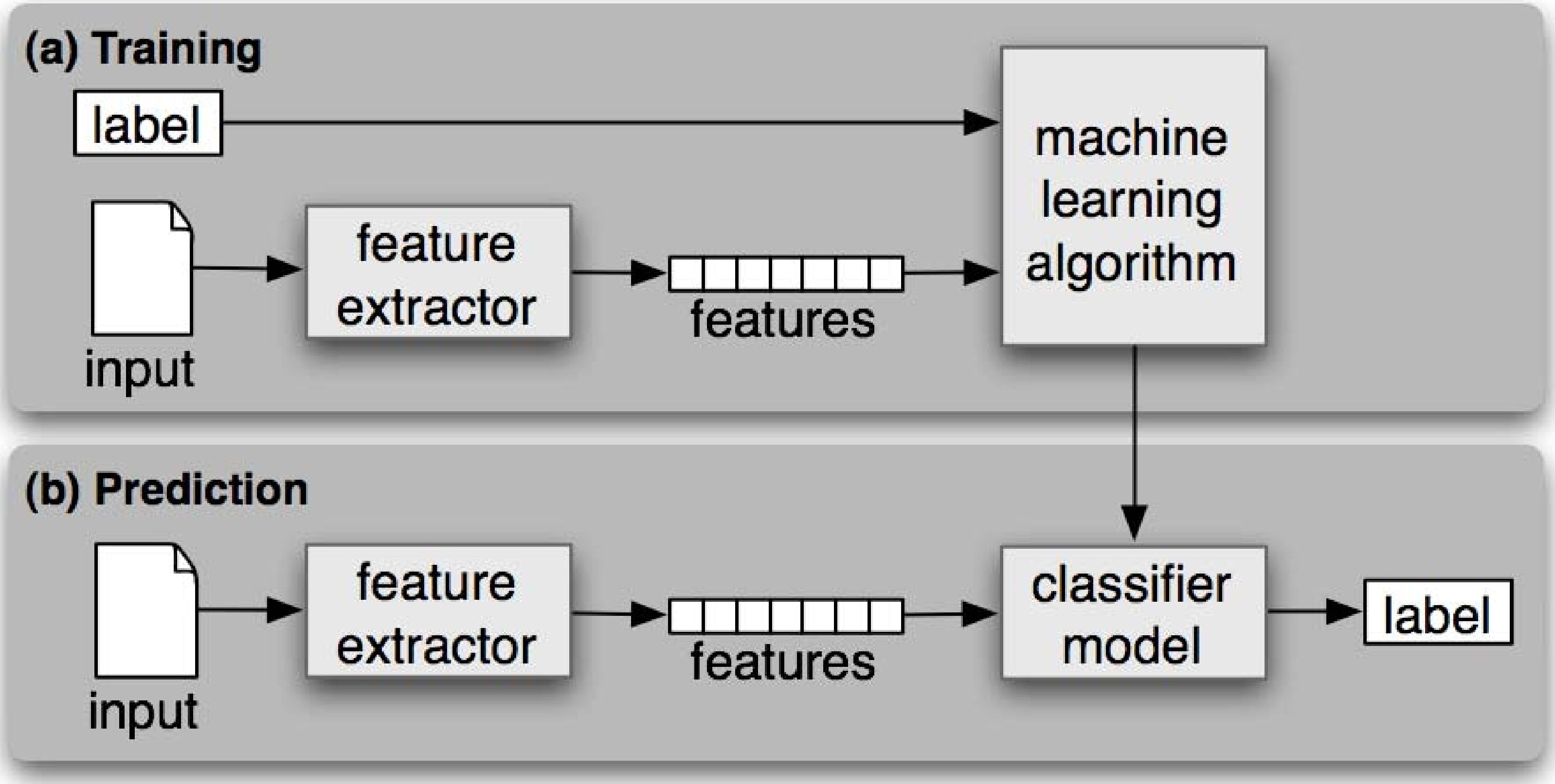
- Binary classification: there are only two groups or class labels.
- Multi-class classification: there are more than two class labels.
- Open-class classification/Clustering: labels are not known in advance.
- Regression: the label variable is a continuous value, e.g. [0-1].
- Sequence classification: a list of inputs are jointly classified.

- Anyway, given the data $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ the classification task can be formalized by a classification function (rule) $f(x) = \hat{y}$.

Supervised Classification

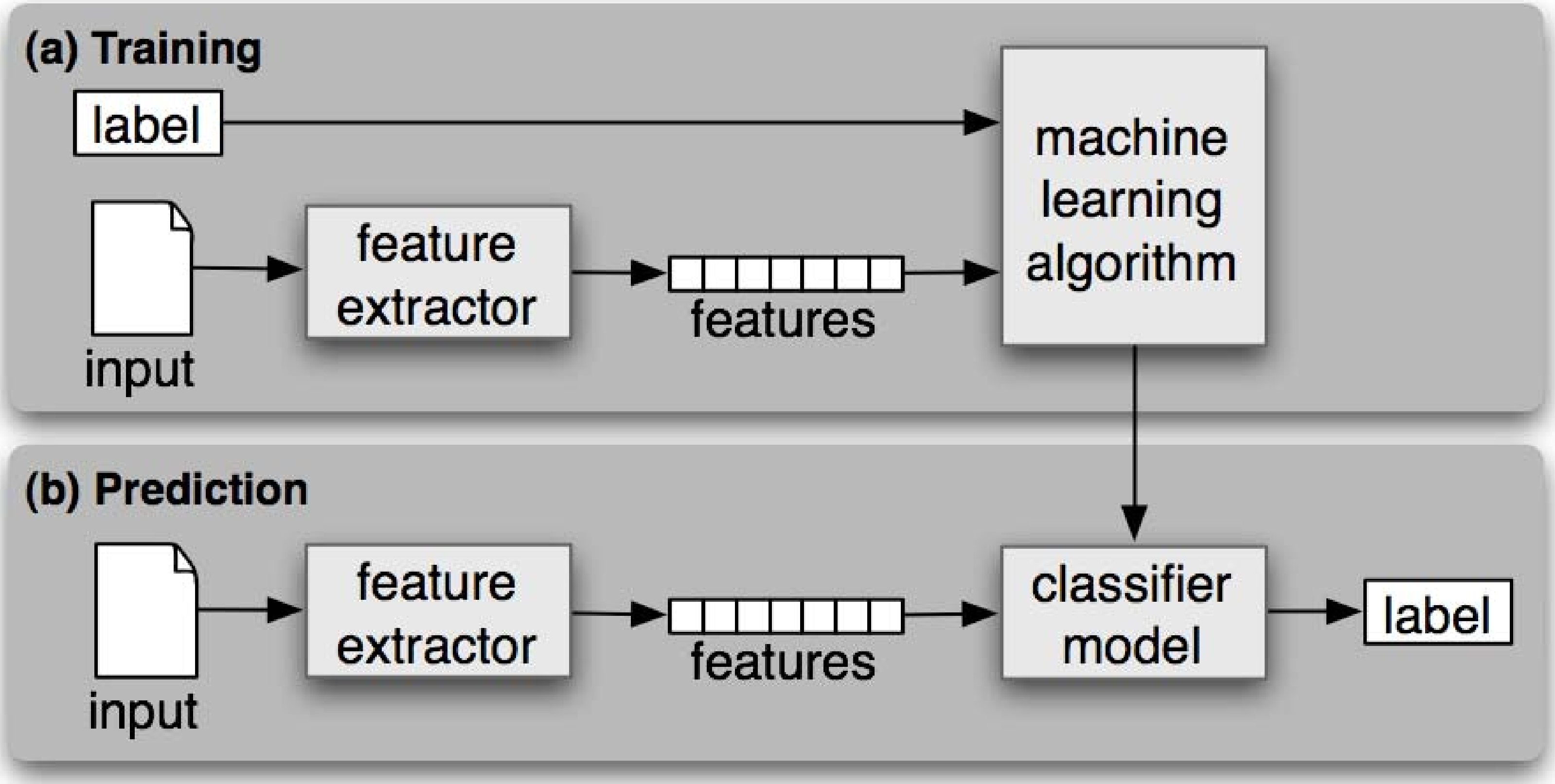
- In supervised classification, a training corpora containing the correct label for each input is available.

Supervised Classification

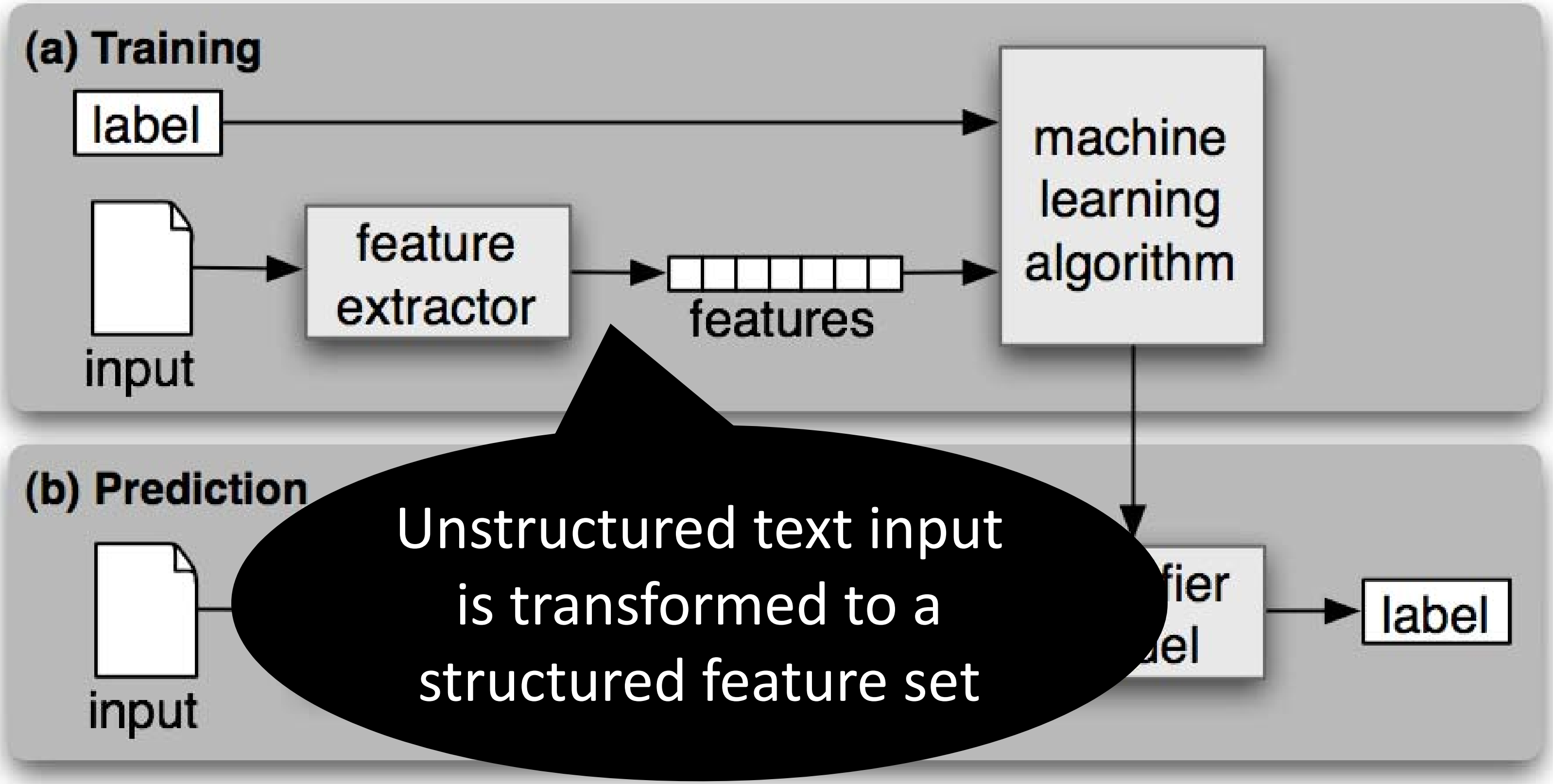


Supervised Classification

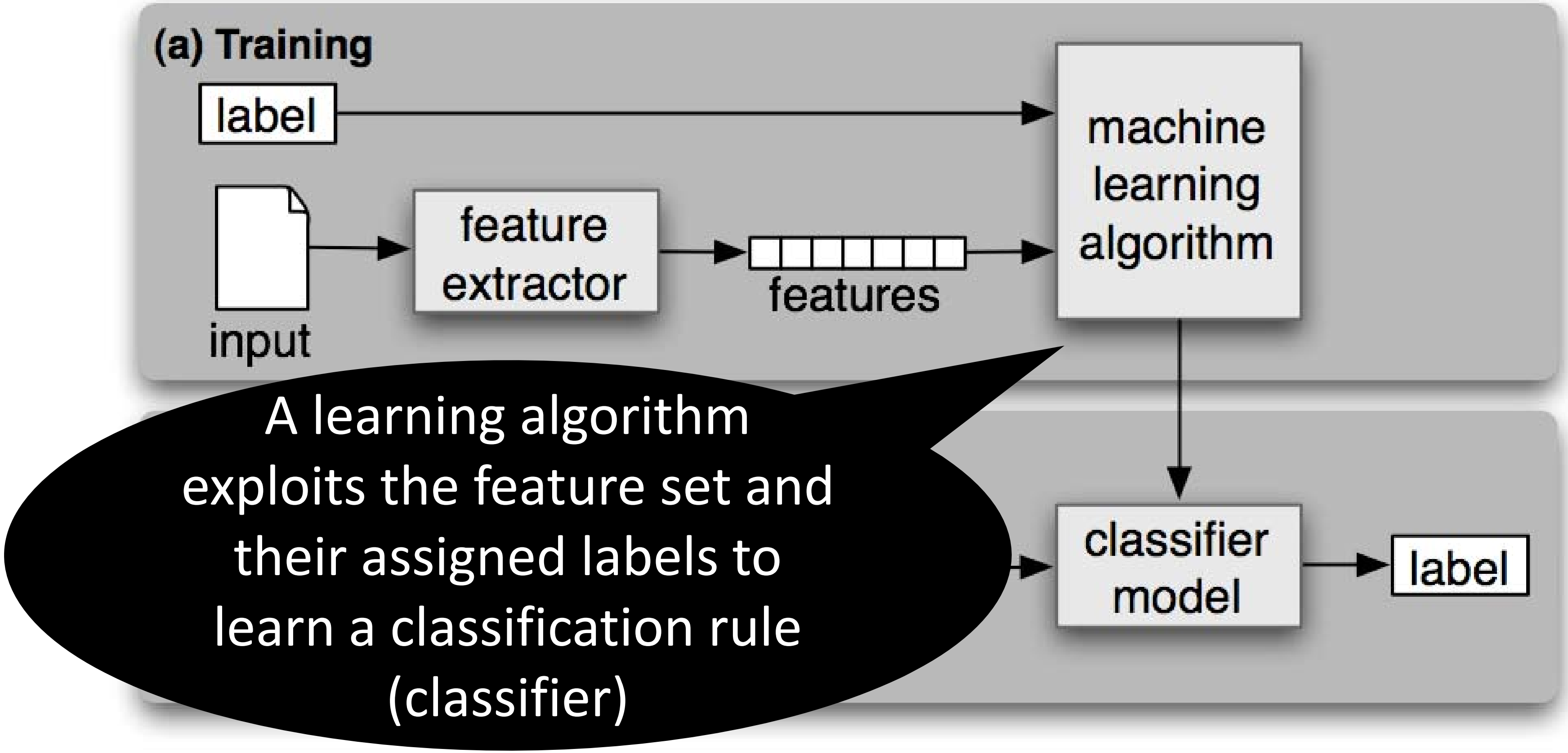
Two Phase of Training and Prediction



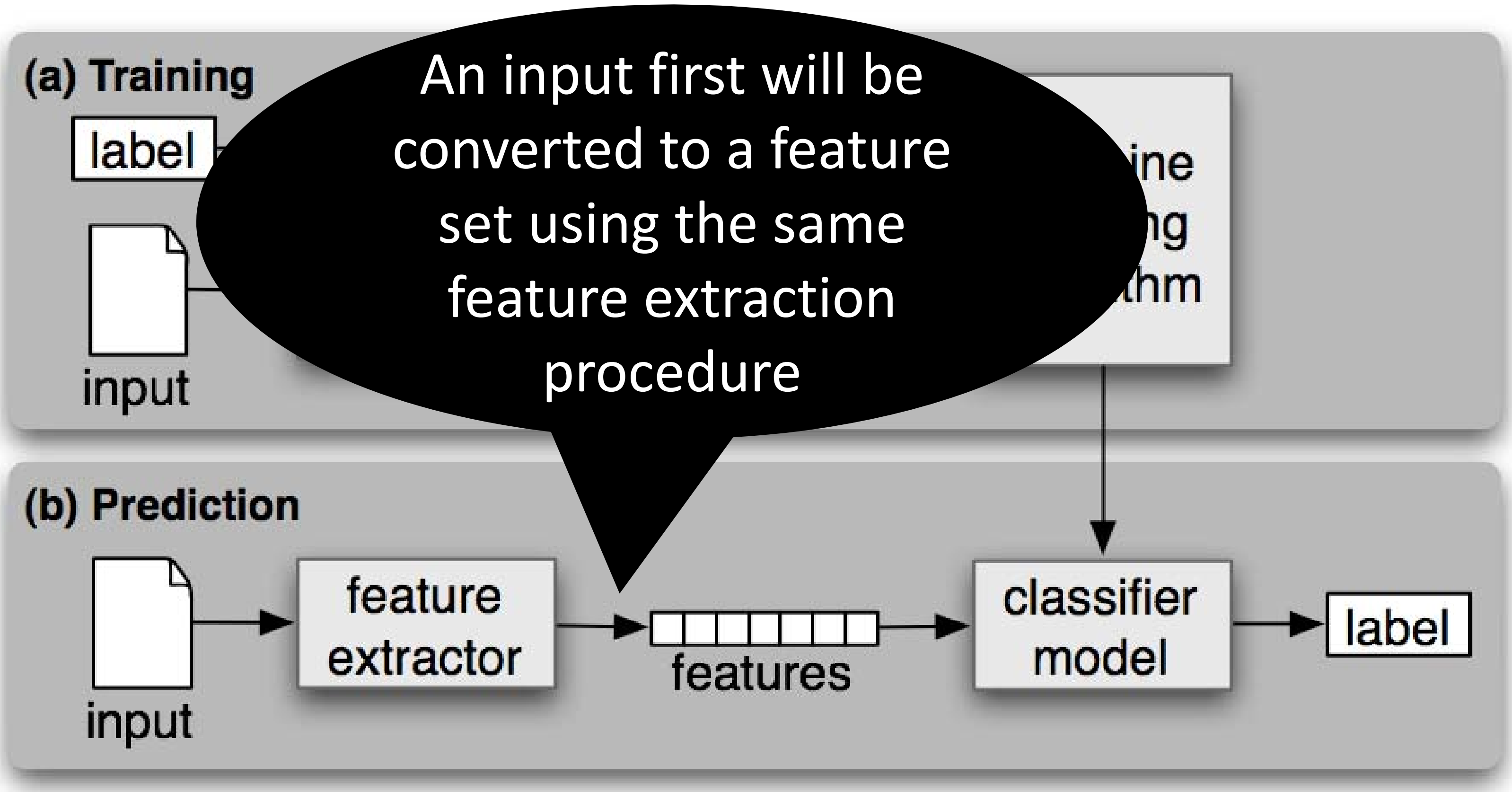
Supervised Classification



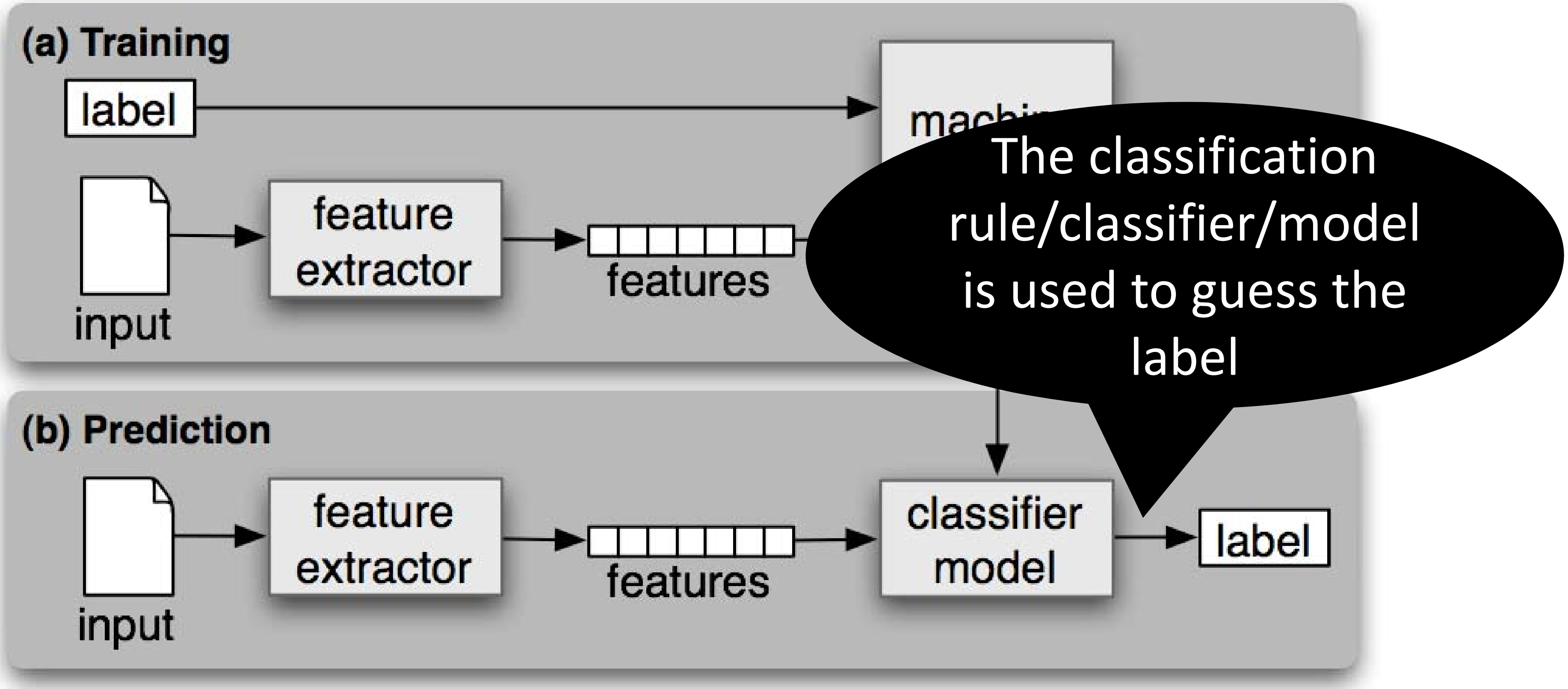
Supervised Classification



Supervised Classification



Supervised Classification



Classification of Classification Techniques

- Based on criteria such as the *employed formalism for encoding the feature set* and *the approach for the representation of the classification rule*, we can identify several classification techniques:
  - Based on probability theory, e.g. using mathematical tools of Bayes' theorem:
    - Naive Bayes Classifier
  - Based on information theory, e.g. using concepts such as mutual information:
    - Maximum Entropy Classifier
  - Based on vector spaces, e.g. using concepts such as distance and similarity:
    - K-Nearest Neighbour Classification
  - Etc.

Classification of Classification Techniques

- In all these methods however you can identify some patterns:
  - There exists a classification rule or a classifier
  - There exists a performance measure for the classifier
  - The learning procedure is often modelled by a *loss function* or *a cost function:*
    - Learning as an optimization problem.
- Our discussion is not intended to be comprehensive, but to give a representative sample of tasks that can be performed with the help of text classifiers.

Name Gender Identification

- Can we guess the gender of a person from his/her name?
  - As examined earlier, English names ending in *a, e* and *i* are likely to be female, while names ending in *k, o, r, s* and *t* are likely to be male.
- Lets build a classifier to model this problem.

Name Gender Identification

- Step 1: decide what features of the input are important, and how to encode those features.

Name Gender Identification

- Step 1: decide what features of the input are important, and how to encode those features.
  - Let’s start by looking at the final letter of a given name.

Name Gender Identification

- Step 1: decide what features of the input are important, and how to encode those features.
  - Let’s start by looking at the final letter of a given name.

```
>>> def gender_features(word):
        return {'last_letter': word[-1]}
```

Name Gender Identification

- Step 1: decide what features of the input are important, and how to encode those features.
  - Let’s start by looking at the final letter of a given name.

```
>>> def gender_features(word):
        return {'last_letter': word[-1]}
>>> gender_features('megamind')
{'last_letter': ‘d'}
```

Name Gender Identification

- Step 1: decide what features of the input are important, and how to encode those features.
  - Let’s start by looking at the final letter of a given name.

```
>>> def gender_features(word):
        return {'last_letter': word[-1]}
>>> gender_features('megamind')
{'last_letter': ‘d'}
```

Name Gender Identification

- Step 1: decide what features of the input are important, and how to encode those features.
  - Let’s start by looking at the final letter of a given name.

```
>>> def gender_features(word):
        return {'last_letter': word[-1]}
>>> gender_features('megamind')
{'last_letter': ‘d'}
```

The feature set is represented by a dictionary!

Name Gender Identification

- Step 1: decide what features of the input are important, and how to encode those features.
  - Let’s start by looking at the final letter of a given name.

```
>>> def gender_features(word):
        return {'last_letter': word[-1]}
>>> gender_features('mega[...]nd')
{'last_letter': ‘d'}
```

Feature names are the keys of the dictionary and usually are shortened!

Name Gender Identification

- Step 1: decide what features of the input are important, and how to encode those features.
  - Let’s start by looking at the final letter of a given name.

```
>>> def gender_features(word):
        return {'last_letter': word[-1]}
>>> gender_features('megamind')
{'last_letter': ‘d'}
```

The feature values are often simple values e.g. a number, Boolean or string value

Name Gender Identification

- Step 1: decide what features of the input are important, and how to encode those features.
  - Let’s start by looking at the final letter of a given name.

```
>>> def gender_features(word):
        return {'last_letter': word[-1]}
>>> gender_features('megamind')
{'last_letter': ‘d'}
```

This is simply a feature extractor

Name Gender Identification

- Step 2: prepare a list of examples and corresponding class labels
    - For this example, let’s use the names dictionary in NTLK.

Name Gender Identification

- Step 2: prepare a list of examples and corresponding class labels
  - For this example, let’s use the names dictionary in NTLK.

```
>>> from nltk.corpus import names
>>> import random
>>> names = ([(name, 'male') for name in names.words('male.txt')] +
      [(name, 'female') for name in names.words('female.txt')])
>>> random.shuffle(names)
>>> featuresets = [(gender_features(n), g) for (n,g) in names]
>>> train_set, test_set = featuresets[500:], featuresets[:500]
```

Name Gender Identification

- Step 2: prepare a list of examples and corresponding class labels
  - For this example, let’s use the names dictionary in NTLK.

```
>>> from nltk.corpus import names
>>> import random
>>> names = ([(name, 'male') for name in names.words('male.txt')] +
      [(name, 'female') for name in names.words('female.txt')])
>>> random.shuffle(names)
>>> featuresets = [(gender_features(n), g) for (n,g) in names]
>>> train_set, test_set = featuresets[500:], featuresets[:500]
```

Pre-process text and create randomly generated test and train data

Name Gender Identification

- Step 3: build a classifier from the feature set.
  - For this example, we skip a few details and directly use naïve Bayes classifer.

```
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
```

Name Gender Identification

- Step 3: build a classifier from the feature set.
  - For this example, we skip a few details and directly use naïve Bayes classifer.

```
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
```

Do what is needed to be done and develop a classifier!

Name Gender Identification

- Step 3: build a classifier from the feature set.
  - For this example, we skip a few details and directly use naïve Bayes classifer.

```
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> classifier.classify(gender_features('Brian'))
'male'
>>> classifier.classify(gender_features(‘Kathy'))
'female'
```

Name Gender Identification

- Step 3: build a classifier from the feature set.
  - For this example, we skip a few details and directly use naïve Bayes classifer.

```
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
```

Name Gender Identification

- Step 4: evaluate the classifier in a systematic way on a larger quantity of unseen data:
  - In this example, use the **`test_set`**.

Name Gender Identification

- Step 4: evaluate the classifier in a systematic way on a larger quantity of unseen data:
  - In this example, use the **test_set.**

```
>>> print nltk.classify.accuracy(classifier, test_set)
0.774
```

Name Gender Identification

- Also, you can have a look at the most informative/discriminative features:

```
>>> classifier.show_most_informative_features(5)
Most Informative Features
last_letter = 'a' female : male = 34.5 : 1.0
last_letter = 'k' male : female = 29.7 : 1.0
last_letter = 'f' male : female = 26.5 : 1.0
last_letter = 'v' male : female = 10.5 : 1.0
last_letter = 'p' male : female = 10.5 : 1.0
```

Name Gender Identification

- Also, you can have a look at the most informative/discriminative features:

```
>>> classifier.show_most_informative_features(5)
Most Informative Features
last_letter = 'a' female : male = 34.5 : 1.0
last_letter = 'k' male : female = 29.7 : 1.0
last_letter = 'f' male : female = 26.5 : 1.0
last_letter = 'v' male : female = 10.5 : 1.0
last_letter = 'p' male : female = 10.5 : 1.0
```

Names in the training set that end in "a" are 34.5 times more likely to be female than male.

Name Gender Identification

- Also, you can have a look at the most informative/discriminative features:

```
>>> classifier.show_most_informative_features(5)
Most Informative Features
last_letter = 'a' female : male = 34.5 : 1.0
last_letter = 'k' male : female = 29.7 : 1.0
last_letter = 'f' male : female = 26.5 : 1.0
last_letter = 'v' male : female = 10.5 : 1.0
last_letter = 'p' male : female = 10.5 : 1.0
```

But, names that end in "k" are 29.7 times more likely to be male than female.

Excersise

- Modify the `gender_features()` function in order to add several other features such as length of names, the first letter, etc.
  - Use the function `nltk.classify.apply_features` to avoid storing very large list of features:

```
>>> from nltk.classify import apply_features
>>> train_set = apply_features(gender_features, names[500:])
>>> test_set = apply_features(gender_features, names[:500])
```

Excersise

- Modify the gender_features(… eral other features such as lengt…
  - Use the function nltk.clas… large list of features:

Use the LazyMap class to construct a lazy list-like object that is analogous to map(feature_func, toks).

```
>>> from nltk.classify import apply_features
>>> train_set = apply_features(gender_features, names[500:])
>>> test_set = apply_features(gender_features, names[:500])
```

Choosing The Right Features

- Selecting relevant features and their proper representation is one of the most important task in the development of a classifier.
- Feature Selection is thus a common term in machine learning.

Choosing The Right Features

- Selecting relevant features and their proper representation is one of the most important task in the development of a classifier.
- Feature Selection is thus a common term in machine learning.
- Typically, feature extractors are built through a trial-and-error process, guided by some intuition of what can be important.

Choosing The Right Features

Of course there are some reports/publications/experiments that can help!

- Selecting relevant features and thei
  the most important task in the develo
- Feature Selection is thus a common term in machine ing.
- Typically, feature extractors are built through a trial-and-error process, guided by some intuition of what can be important.

Choosing The Right Features

- Selecting relevant features and their proper representation is one of the most important task in the development of a classifier.
- Feature Selection is thus a common term in machine learning.
- Typically, feature extractors are built through a trial-and-error process, guided by some intuition of what can be important.
- It is common to start with a greedy "kitchen sink" approach:
  - First, Generate all the features that you can think of;
  - Then, check and see which one is actually useful/discriminative

Choosing The Right Features

- It is common to start with a greedy "kitchen sink" approach:
  - First, Generate all the features that you can think of;
  - Then, check and see which one is actually useful/discriminative

```
def gender_features2(name):
    features = {}
    features["firstletter"] = name[0].lower()
    features["lastletter"] = name[-1].lower()
    for letter in 'abcdefghijklmnopqrstuvwxyz':
        features["count(%s)" % letter] = name.lower().count(letter)
        features["has(%s)" % letter] = (letter in name.lower())
    return features
```

Choosing The Right Features

- It is common to start with a greedy "kitchen sink" approach:
  - First, Generate all the features that you can think of;

```
def gender_features2(name):
        features = {}
        features["firstletter"] = name[0].lower()
        features["lastletter"] = name[-1].lower()
        for letter in 'abcdefghijklmnopqrstuvwxyz':
                features["count(%s)" % letter] = name.lowe
                features["has(%s)" % letter] = (lett
        return features
```

Overfitting Problem!

Choosing The Right Features

- The greedy method of generating a lot of features, however, comes with certain limitations:
  - Overfitting Problem:
    - The larger the number of features, the higher the chance of relying on idiosyncrasies of training data, specially when the size of training data is small.
    - In this case, the generated classifier don't generalize well to new examples.

Choosing The Right Features

- The greedy method of generating a lot of features, however, comes with certain limitations:
  - Overfitting Problem:
    - The larger the number of features, the higher the chance of relying on idiosyncrasies of training data, specially when the size of training data is small.
    - In this case, the generated classifier don't generalize well to new examples.

```
>>> featuresets = [(gender_features2(n), g) for (n,g) in names]
>>> train_set, test_set = featuresets[500:], featuresets[:500]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print nltk.classify.accuracy(classifier, test_set)
0.764
```

Choosing The Right Features

- The greedy method of generating a lot of features, however, comes with certain limitations:
  - Overfitting Problem:
    - The larger the number of fea ... iosyncrasies of training data, specially wh
    - In this case, the generat

```
>>> featuresets = ... es]
>>> train_set, tes ... 0]
>>> classifier =
>>> print nl
0.764
```

Using the new feature set, compared to the earlier result of 0.774, the performance has dropped by almost 1%!

Choosing The Right Features

- The greedy method of generating a lot of features, however, comes with certain limitations:
  - Overfitting Problem:
    - The larger the number of features, the higher the chance of relying on idiosyncrasies of training data, specially when the size of training data is small.
    - In this case, the generated classifier don't generalize well to new examples.

```
>>> featuresets = [(gender_features2(n), g) for (n,g) in names]
>>> train_set, test_set = featuresets[500:], featuresets[:500]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print nltk.classify.accuracy(classifier, test_set)
0.764
```

- So, we need to limit the number of features.

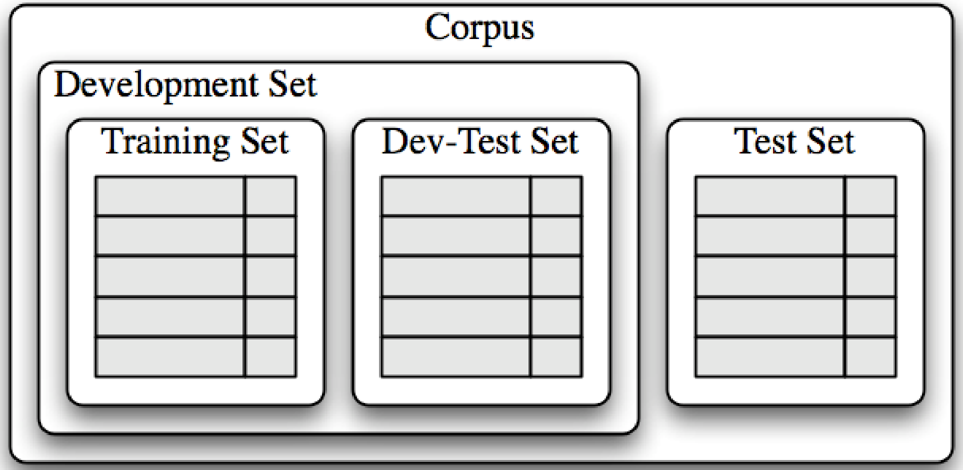
Choosing The Right Features

- One effective method for refining the feature set is error analysis.
  - First, we select a development training set, containing the corpus data for creating the model.
  - This development set is then subdivided into two subsets: the training set and the development-test set.

Choosing The Right Features

- One effective method for refining the feature set is error analysis.
  - First, we select a development training set, containing the corpus data for creating the model.
  - This development set is then subdivided into two subsets: the training set and the development-test set.

```
>>> train_names = names[1500:]
>>> devtest_names = names[500:1500]
>>> test_names = names[:500]
```

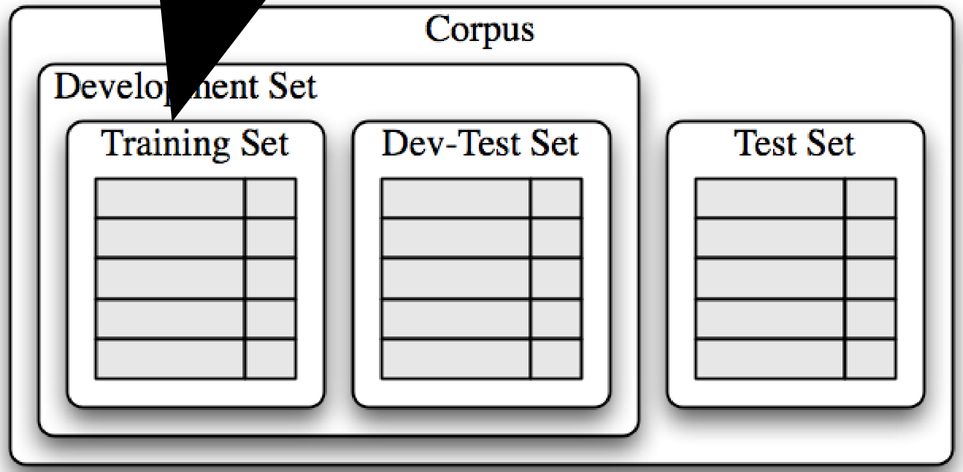


Choosing The Right Features

- One effective method for refining t… …alysis.
  - First, we select a development traini… …data for creating the model.
  - This development set is then subdivided in… …e training set and the development-test set.

Use the training set to develop a classifier

```
>>> train_names = names[1500:]
>>> devtest_names = names[500:1500]
>>> test_names = names[:500]
```

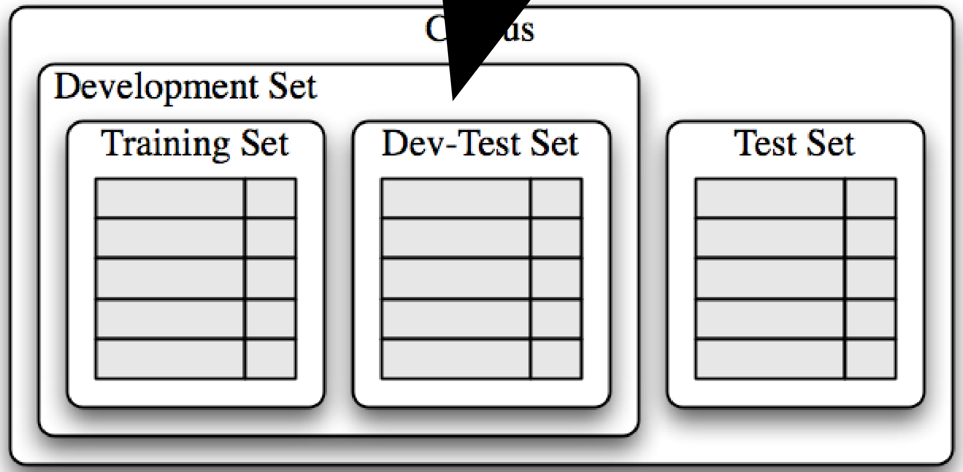


Choosing The Right Features

- One effective method for refining the feature
  - First, we select a development training set, conta
creating the model.
  - This development set is then subdivided into two subsets. set and the development-test set.

Use the Dev-Test set to do the error analysis.

```
>>> train_names = names[1500:]
>>> devtest_names = names[500:1500]
>>> test_names = names[:500]
```

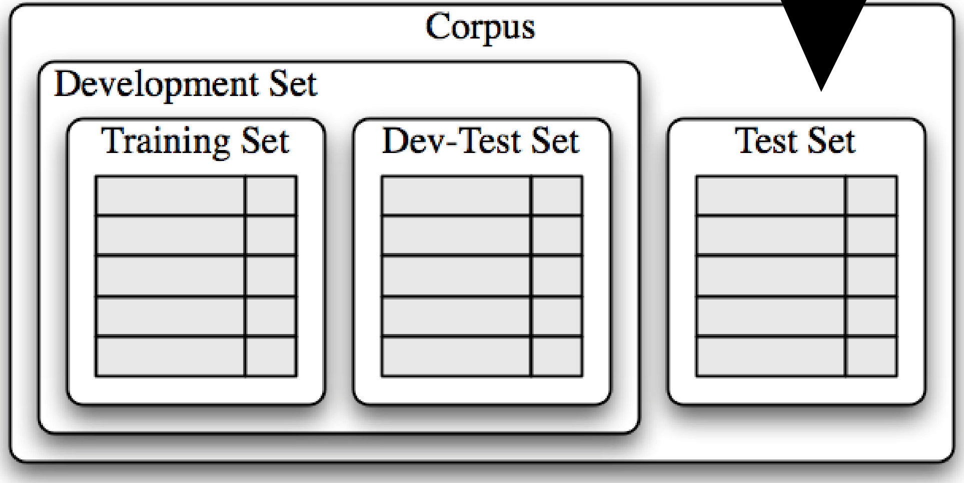


Choosing The Right Features

- One effective method for refining the feature set is error
  - First, we select a development training set, containing the cor creating the model.
  - This development set is then subdivided into two subsets: the trainin the development-test set.

Finally, use the test set to do the evaluation

```
>>> train_names = names[1500:]
>>> devtest_names = names[500:1500]
>>> test_names = names[:500]
```



Choosing The Right Features

```
>>> train_set = [(gender_features(n), g) for (n,g) in train_names]
>>> devtest_set = [(gender_features(n), g) for (n,g) in devtest_names]
>>> test_set = [(gender_features(n), g) for (n,g) in test_names]
```

Perform feature extractions on these sets

Choosing The Right Features

```
>>> train_set = [(gender_features(n), g) for (n,g) in train_names]
>>> devtest_set = [(gender_features(n), g) for (n,g) in devtest_names]
>>> test_set = [(gender_features(n), g) for (n,g) in test_names]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
```

Develop the classifier

Choosing The Right Features

```
>>> train_set = [(gender_features(n), g) for (n,g) in train_names]
>>> devtest_set = [(gender_features(n), g) for (n,g) in devtest_names]
>>> test_set = [(gender_features(n), g) for (n,g) in test_names]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print nltk.classify.accuracy(classifier, devtest_set)
 0.755
```

Evaluate it on the **devtest_set**

Choosing The Right Features

- Use the **`devtest_set`** to inspect errors!

First, make an inventory of difficult entries.

```
>>> errors = []
>>> for (name, tag) in devtest_names:
      guess = classifier.classify(gender_features(name))
      if guess != tag:
             errors.append( (tag, guess, name) )
```

Choosing The Right Features

- Use the **devtest_set** to inspect errors!

Now, manually inspect errors and see if you recon any patterns!

```
>>> for (tag, guess, name) in sorted(errors):
      print 'correct=%-8s guess=%-8s name=%-30s' % (tag, guess, name)

correct=female guess=male name=Cindelyn
correct=female guess=male name=Katheryn
correct=female guess=male name=Kathryn
...
```

Choosing The Right Features

- Use the **`devtest_set`** to inspect errors!

Now, manually inspect errors and see if you recon any patterns!

```
>>> for (tag, guess, name) in sorted(errors):
        print 'correct=%-8s guess=%-8s name=%-30s' % (tag, guess, name)

correct=female guess=male name=Cindelyn
correct=female guess=male name=Katheryn
correct=female guess=male name=Kathryn
...
```

Choosing The Right Features

- Use the **`devtest_set`** to inspect errors!

```
>>> for (tag, guess, name) in sorted(errors):
        print 'correct=%-8s guess=%-8s name=%-30s' % (tag, guess, name)

correct=female guess=male name=Cindelyn
correct=female guess=male name=Katheryn
correct=female guess=male name=Kathryn
...
```

adjust feature extractor to include features for two-letter suffixes

Choosing The Right Features

- Use the **`devtest_set`** to inspect errors!
- Amend the feature extraction process!

Choosing The Right Features

- Use the **devtest_set** to inspect errors!
- Amend the feature extraction process!

```
>>> def gender_features(word):
    return {'suffix1': word[-1:], 'suffix2': word[-2:]}
```

Choosing The Right Features

- Use the **`devtest_set`** to inspect errors!
- Amend the feature extraction process!
- Develop a new model and test the new feature set!

Choosing The Right Features

- Use the **devtest_set** to inspect errors!
- Amend the feature extraction process!
- Develop a new model and test the new feature set!

```
>>> train_set = [(gender_features(n), g) for (n,g) in train_names]
>>> devtest_set = [(gender_features(n), g) for (n,g) in devtest_names]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print nltk.classify.accuracy(classifier, devtest_set)
0.782
```

Choosing The Right Features

- Use the **`devtest_set`** to inspect errors!
- Amend the feature extraction process!
- Develop a new model and test the new feature set!
- Repeat the steps listed above!

Choosing The Right Features

- Use the **`devtest_set`** to inspect errors!
- Amend the feature extraction process!
- Develop a new model and test the new feature set!
- Repeat the steps listed above!
  - Important note: each time the error analysis procedure is repeated, select a different dev-test/training data to avoid over-fitting!

Choosing The Right Features

- Use the **`devtest_set`** to inspect errors!
- Amend the feature extraction process!
- Develop a new model and test the new feature set!
- Repeat the steps listed above!
  - Important note: each time the error analysis procedure is repeated, select a different dev-test/training data to avoid over-fitting!
- Test your model on the test set, once you are done with the development procedure.

Document Classification

- We can use corpora of documents that are labelled with categories to develop a document classifier.
- The classifiers then can be used to automatically tag new documents with appropriate category labels.

Document Classification

- We can use corpora of documents that are labelled with categories to develop a document classifier.
- The classifiers then can be used to automatically tag new documents with appropriate category labels.
- The procedure is similar to the previous example:
  - First, construct a list of documents, labelled with the appropriate categories.
  - Second, define a feature extractor for documents.
  - Third, apply feature extraction and develop a classifier.

Document Classification

- Step 1: prepare list of labelled documents

Use **movie_reviews** dataset (each review is labelled either as positive or negative)

```
>>> from nltk.corpus import movie_reviews
>>> documents = [(list(movie_reviews.words(fileid)), category)
        for category in movie_reviews.categories()
        for fileid in movie_reviews.fileids(category)]
>>> random.shuffle(documents)
```

Document Classification

- Step 2: define features

Use the bag-of-words hypothesis (only 2000 most frequent words)

```
>>> all_words = nltk.FreqDist(w.lower() for w in movie_reviews.words())
>>> word_features = all_words.keys()[:2000]

>>> def document_features(document):
       document_words = set(document)
       features = {}
      for word in word_features:
            features['contains(%s)' % word] = (word in document_words)
      return features
```

Document Classification

- Step 3: develop a classifier

Use the bag-of-words hypothesis (only 2000 most frequent words)

```
>>> featuresets = [(document_features(d), c) for (d,c) in documents]
>>> train_set, test_set = featuresets[100:], featuresets[:100]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
```

Document Classification

- Step 3: develop a classifier

Use the bag-of-words hypothesis (only 2000 most frequent words)

```
>>> featuresets = [(document_features(d), c) for (d,c) in documents]
>>> train_set, test_set = featuresets[100:], featuresets[:100]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)

>>> print nltk.classify.accuracy(classifier, test_set)
0.81
>>> lassifier.show_most_informative_features(5)
Most Informative Features
      contains(outstanding) = True pos : neg = 11.1 : 1.0
      contains(seagal) = True neg : pos = 7.7 : 1.0
      contains(wonderfully) = True pos : neg = 6.8 : 1.0
      ...
```

Exercise

- Enhance the document classification by enhancing the feature extraction process, e.g. get rid of stop words!

Part-of-Speech Tagging using Decision trees

- Instead of handcrafted regular expressions for part-of-speech tagging (remember from last session?!), lets use a decision tree!
- Exact same procedure:
  - Prepare data;
  - Define features;
  - Develop the model.

Part-of-Speech Tagging using Decision trees

- Instead of handcrafted regular expressions for part-of-speech tagging (remember from last session?!), lets use a decision tree!
- Exact same procedure:
  - Prepare data;
  - Define features;
  - Develop the model.
- Features here are suffixes that appear at the end of words.

Part-of-Speech Tagging using Decision trees

```
>>> from nltk.corpus import brown
>>> suffix_fdist = nltk.FreqDist()
>>> for word in brown.words():
        word = word.lower()
        suffix_fdist[word[-1:]]+=1
        suffix_fdist[word[-2:]]+=1
        suffix_fdist[word[-3:]]+=1
>>> common_suffixes = [seq[0] for seq
                      in suffix_fdist. most_common(n=100)]
```

Let’s find the top 100 common suffixes!

Part-of-Speech Tagging using Decision trees

```
>>> from nltk.corpus import brown
>>> suffix_fdist = nltk.FreqDist()
>>> for word in brown.words():
      word = word.lower()
      suffix_fdist[word[-1:]]+=1
      suffix_fdist[word[-2:]]+=1
      suffix_fdist[word[-3:]]+=1
>>> common_suffixes = [seq[0] for seq
                   in suffix_fdist. most_common(n=100)]
>>> print common_suffixes
['e', ',', '.', 's', 'd', 't', 'he', 'n', 'a',...]
```

Part-of-Speech Tagging using Decision trees

- A feature extraction function using the extracted suffixes:

```
>>> def pos_features(word):
        features = {}
        for suffix in common_suffixes:
              features['endswith(%s)' %suffix] =\
                           word.lower().endswith(suffix)
        return features
```

Part-of-Speech Tagging using Decision trees

- Now, apply the feature extraction and build a classifier

```
>>> tagged_words = brown.tagged_words(categories='news')
>>> featuresets = [(pos_features(n), g) for (n,g) in tagged_words]
```

Part-of-Speech Tagging using Decision trees

- Now, apply the feature extraction and build a classifier

```
>>> tagged_words = brown.tagged_words(categories='news')
>>> featuresets = [(pos_features(n), g) for (n,g) in tagged_words]

>>> size = int(len(featuresets) * 0.1)
>>> train_set, test_set = featuresets[size:], featuresets[:size]
```

Part-of-Speech Tagging using Decision trees

- Now, apply the feature extraction and build a classifier

```
>>> tagged_words = brown.tagged_words(categories='news')
>>> featuresets = [(pos_features(n), g) for (n,g) in tagged_words]

>>> size = int(len(featuresets) * 0.1)
>>> train_set, test_set = featuresets[size:], featuresets[:size]

>>> classifier = nltk.DecisionTreeClassifier.train(train_set)
>>> nltk.classify.accuracy(classifier, test_set)
0. 6270512182993535
```

Sequence Classification

- Some of the classification tasks are related to each other, i.e. to solve a problem we have to make a chain of decisions:
  - e.g. for Part-of-Speech tagging, choosing a PoS tag for each word will affect the decision for choosing the next one!

Sequence Classification

- Some of the classification tasks are related to each other, i.e. to solve a problem we have to make a chain of decisions:
  - e.g. for Part-of-Speech tagging, choosing a PoS tag for each word will affect the decision for choosing the next one!
- A vibrant research community works on this problem:
  - Markov Chain and the Hidden Markov Model (HMM)
  - Maximum Entropy Markov Model (MEMM)
  - Conditional Random Field
  - etc.

Sequence Classification

- Some of the classification tasks are related to each other, i.e. to … a problem we have to make a chain of decisi…
  - e.g. for Part-of-Speech … choosing a PoS … the decision for choosin…
- A vibrant research comm…
  - Mar…
  - Maximu…
  - Conditional Ra…
  - etc.

Check out
Linguistic Structure Prediction
by Noah Smith!
[Slides: http://lxmls.it.pt/2014/lxmls.7-24-14.pdf](http://lxmls.it.pt/2014/lxmls.7-24-14.pdf)

Sequence Classification

- One strategy for sequence labelling is to find the most likely class label for the first input, then to use that answer to help find the best label for the next input.
  - Repeat the process until all of the inputs have been labelled
  - Similar to n-gram tagging from Chapter 5, if you remember?!
  - Sounds like dynamic programming!

Sequence Classification

- One strategy for sequence labelling is to find the most likely class label for the first input, then to use that answer to help find the best label for the next input.
  - Repeat the process until all of the inputs have been labelled
  - Similar to n-gram tagging from Chapter 5, if you remember?!
  - Sounds like dynamic programming!
- One of the main differences here is the implementation of feature extraction:
  - We must enable our feature extractor to take a history argument.

Sequence Classification

- One strategy for sequence labelling is to find the most likely class label for the first input, then to use that answer to help find the best label for the next input.
  - Repeat the process until all of the inputs have been labelled
  - Similar to n-gram tagging from Chapter 5, if you remember?!
  - Sounds like dynamic programming!
- One of the main differences here is the implementation of feature extraction:
  - We must enable our feature extractor to take a history argument.

Sequence Classification

```
def pos_features(sentence, i, history):
      features = {"suffix(1)": sentence[i][-1:],
                  "suffix(2)": sentence[i][-2:],
                  "suffix(3)": sentence[i][-3:]}
      if i == 0:
            features["prev-word"] = "<START>“
            features["prev-tag"] = "<START>"
      else:
            features["prev-word"] = sentence[i-1]
            features["prev-tag"] = history[i-1]
return features
```

Sequence Classification

```
def pos_features(sentence, i, history):
      features = {"suffix(1)": sentence[i][-1:],
                  "suffix(2)": sentence[i][-2:],
                  "suffix(3)": sentence[i][-3:]}
      if i == 0:
            features["prev-word"] = "<START>“
            features["prev-tag"] = "<START>"
      else:
            features["prev-word"] = sentence[i-1]
            features["prev-tag"] = history[i-1]
return features
```

I can remember a few things

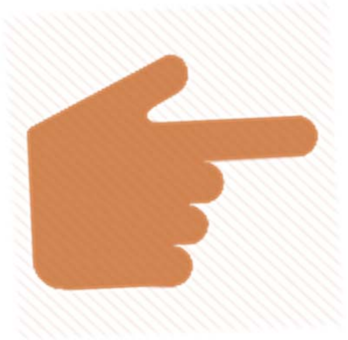
Sequence Classification

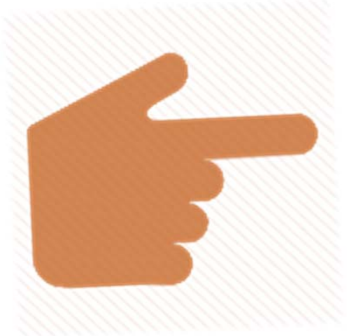
```
class ConsecutivePosTagger(nltk.TaggerI):
      def __init__(self, train_sents):
            train_set = []
            for tagged_sent in train_sents:
                  untagged_sent = nltk.tag.untag(tagged_sent)
            history = []
            for i, (word, tag) in enumerate(tagged_sent):
                  featureset = pos_features(untagged_sent, i, history)
                  train_set.append( (featureset, tag) )
                  history.append(tag)
            self.classifier = nltk.NaiveBayesClassifier.train(train_set)

      def tag(self, sentence):
            history = []
            for i, word in enumerate(sentence):
                  featureset = pos_features(sentence, i, history)
                  tag = self.classifier.classify(featureset)
                  history.append(tag) return zip(sentence, history)
```

Sequence Classification

```
class ConsecutivePosTagger(nltk.TaggerI):
    def __init__(self, train_sents):
        train_set = []
        for tagged_sent in train_sents:
            untagged_sent = nltk.tag.untag(tagged_sent)
        history = []
        for i, (word, tag) in enumerate(tagged_sent):
            featureset = pos_features(untagged_sent, i, history)
            train_set.append( (featureset, tag) )
            history.append(tag)
        self.classifier = nltk.NaiveBayesClassifier.train(train_set)

    def tag(self, sentence):
        history = []
        for i, word in enumerate(sentence):
            featureset = pos_features(sentence, i, history)
            tag = self.classifier.classify(featureset)
            history.append(tag) return zip(sentence, history)
```

Sequence Classification





```
class ConsecutivePosTagger(nltk.TaggerI):
      def __init__(self, train_sents):
            train_set = []
            for tagged_sent in train_sents:
                  untagged_sent = nltk.tag.untag(tagged_sent)
            history = []
            for i, (word, tag) in enumerate(tagged_sent):
                  featureset = pos_features(untagged_sent, i, history)
                  train_set.append( (featureset, tag) )
                  history.append(tag)
            self.classifier = nltk.NaiveBayesClassifier.train(train_set)

      def tag(self, sentence):
            history = []
            for i, word in enumerate(sentence):
                  featureset = pos_features(sentence, i, history)
                  tag = self.classifier.classify(featureset)
                  history.append(tag) return zip(sentence, history)
```

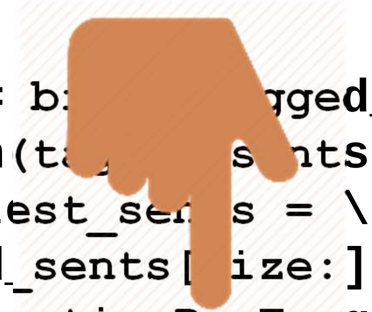
Sequence Classification

```
>>> tagged_sents = brown.tagged_sents(categories='news')
>>> size = int(len(tagged_sents) * 0.1)
>>> train_sents, test_sents = \
        tagged_sents[size:], tagged_sents[:size]
```

Sequence Classification

```
>>> tagged_sents = brown.tagged_sents(categories='news')
>>> size = int(len(tagged_sents) * 0.1)
>>> train_sents, test_sents = \
            tagged_sents[size:], tagged_sents[:size]
>>> tagger = ConsecutivePosTagger(train_sents)
```

Sequence Classification



```
>>> tagged_sents = brown.tagged_sents(categories='news')
>>> size = int(len(tagged_sents) * 0.1)
>>> train_sents, test_sents = \
            tagged_sents[size:], tagged_sents[:size]
>>> tagger = ConsecutivePosTagger(train_sents)
>>> print(tagger.evaluate(test_sents))
0.79796012981
```

Other examples

- The NLTK book comes with several interesting examples; each example targets a specific concept:
  - **Sentence Segmentation**
  - **Identifying Dialogue Act Types**
  - **Recognizing Textual Entailment**

Scaling Up to Large Datasets

- “Python provides an excellent environment for performing basic text processing and feature extraction. However, it is not able to perform the numerically intensive calculations required by machine learning methods nearly as quickly as lower-level languages such as C.”

Scaling Up to Large Datasets

- “Python provides an excellent environment for performing basic text processing and feature extraction. However, it is not able to perform the numerically intensive calculations required by machine learning methods nearly as quickly as lower-level languages such as C.”
- “Thus, if you attempt to use the pure-Python machine learning implementations (such as nltk.NaiveBayesClassifier) on large datasets, you may find that the learning algorithm takes an unreasonable amount of time and memory to complete.”

Scaling Up to Large Datasets

- “Python provides a[redacted]xcellent en[redacted]rformi[redacted] basic text processing and featu[redacted] to perform the numerically inten[redacted]ng methods n[redacted] C.”
- “Thus, if you att[redacted]ning implem[redacted]ge datasets, you may fin[redacted]unreas[redacted]ble amount of ti[redacted]

However, don’t blame Python on your errors in Programming!

Scaling Up to Large Datasets

- Please double check programming patterns in Chapter 4:
  - **Functions as Arguments**
  - **Accumulative Functions**
  - **Higher-Order Functions!**

We continue with classification and learning techniques next session!

- Before we finish:
  - Do you know about FLASK?
    - http://flask.pocoo.org/
  - What do you think of FLASK + jquery?
  - Do you have any suggestion other than FLASK?!