Elements Of Data Science - F2021

Week 10: NLP, Sentiment Analysis and Topic Modeling

11/22/2021

TODOs

- Readings:
  - PDSH 5.11 k-Means
  - [Recommended] PML Chapter 11: Working with Unlabeled Data - Clustering Analysis except for last section on DBScan
  - [Optional] Data Science From Scratch Chap 22: Recommender Systems

- HW3, Due **Tues Nov 23rd 11:59pm**
- Answer and submit Quiz 10, **Sunday Nov 28, 11:59pm ET**

Today

- Pipelines
- NLP
- Sentiment Analysis
- Topic Modeling

Questions?

Environment Setup

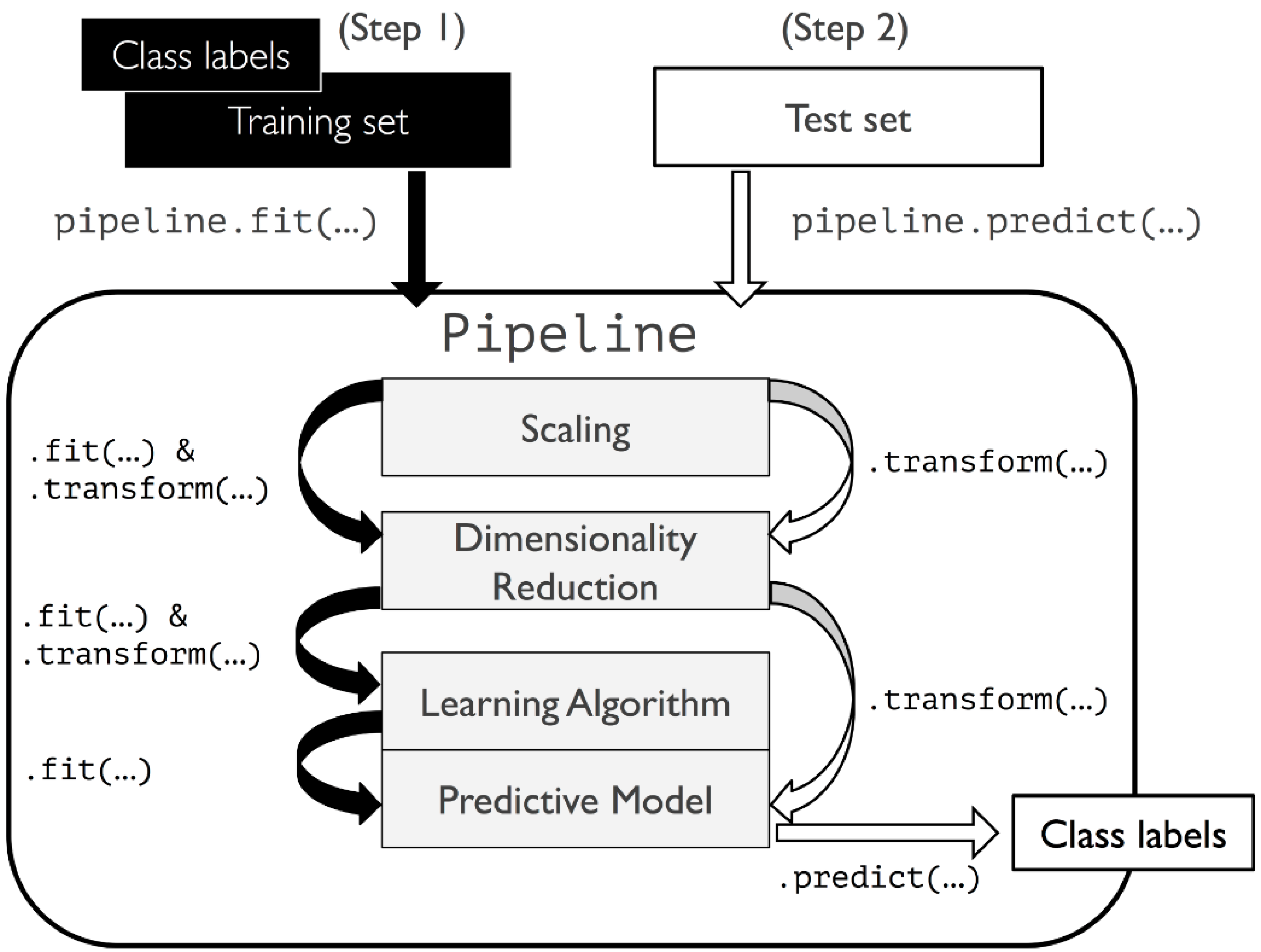
Environment Setup

In [1]:
```
import numpy
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

import warnings
warnings.filterwarnings('ignore')

sns.set_style('darkgrid')
%matplotlib inline
```

Pipelines in sklearn

- Pipelines are wrappers used to string together transformers and estimators
- sequentially apply a series of transforms, eg, `.fit_transform()` and `.transform()`
- followed by a prediction, eg. `.fit()` and `.predict()`

Pipelines in sklearn



From PML

Binary Classification With All Numeric Features Setup

Binary Classification With All Numeric Features Setup

In [2]:

```
# Example from PML - scaling > feature extraction > classification
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
bc = load_breast_cancer()
X_bc,y_bc = bc['data'],bc['target']
X_bc_train,X_bc_test,y_bc_train,y_bc_test = train_test_split(X_bc,
                                                             y_bc,
                                                             test_size=0.2,
                                                             stratify=y_bc,
                                                             random_state=123)

# all real valued features
X_bc_train[:1]
```

Out[2]:

```
array([[1.094e+01, 1.859e+01, 7.039e+01, 3.700e+02, 1.004e-01, 7.460e-02,
        4.944e-02, 2.932e-02, 1.486e-01, 6.615e-02, 3.796e-01, 1.743e+00,
        3.018e+00, 2.578e+01, 9.519e-03, 2.134e-02, 1.990e-02, 1.155e-02,
        2.079e-02, 2.701e-03, 1.240e+01, 2.558e+01, 8.276e+01, 4.724e+02,
        1.363e-01, 1.644e-01, 1.412e-01, 7.887e-02, 2.251e-01, 7.732e-02]])
```

Pipelines in sklearn

Pipelines in sklearn

In [3]:
```python
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression

# Pipeline: list of (name,object) pairs
pipe1 = Pipeline([('scale',StandardScaler()),                   # scale
                 ('pca',PCA(n_components=2)),                   # reduce dimensions
                 ('lr',LogisticRegression(solver='saga',
                                          max_iter=1000,
                                          random_state=123)),  # classifier
                ])

pipe1.fit(X_bc_train,y_bc_train)

print(f'train set accuracy: {pipe1.score(X_bc_train,y_bc_train):0.3f}')
print(f'test set accuracy : {pipe1.score(X_bc_test,y_bc_test):0.3f}')
```

```
train set accuracy: 0.956
test set accuracy : 0.956
```

Pipelines in sklearn

In [3]:
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression

# Pipeline: list of (name,object) pairs
pipe1 = Pipeline([('scale',StandardScaler()),                  # scale
                  ('pca',PCA(n_components=2)),                  # reduce dimensions
                  ('lr',LogisticRegression(solver='saga',
                                           max_iter=1000,
                                           random_state=123)),  # classifier
                 ])

pipe1.fit(X_bc_train,y_bc_train)

print(f'train set accuracy: {pipe1.score(X_bc_train,y_bc_train):0.3f}')
print(f'test set accuracy : {pipe1.score(X_bc_test,y_bc_test):0.3f}')
```

```
train set accuracy: 0.956
test set accuracy : 0.956
```

In [4]:
```
# access pipeline components by name like a dictionary
pipe1['lr'].coef_
```

Out[4]:
```
array([[-2.00439115,  1.11969368]])
```

Pipelines in sklearn

In [3]:
```python
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression

# Pipeline: list of (name,object) pairs
pipe1 = Pipeline([('scale',StandardScaler()),                    # scale
                  ('pca',PCA(n_components=2)),                   # reduce dimensions
                  ('lr',LogisticRegression(solver='saga',
                                           max_iter=1000,
                                           random_state=123)),   # classifier
                ])

pipe1.fit(X_bc_train,y_bc_train)

print(f'train set accuracy: {pipe1.score(X_bc_train,y_bc_train):0.3f}')
print(f'test set accuracy : {pipe1.score(X_bc_test,y_bc_test):0.3f}')
```

```
train set accuracy: 0.956
test set accuracy : 0.956
```

In [4]:
```python
# access pipeline components by name like a dictionary
pipe1['lr'].coef_
```

Out[4]:
```
array([[-2.00439115,  1.11969368]])
```

In [5]:
```python
pipe1['pca'].components_[0]
```

Out[5]:
```
array([0.21777854, 0.08876361, 0.22663097, 0.22043131, 0.14913361,
       0.23954684, 0.25974993, 0.26277752, 0.14518851, 0.06537618,
       0.20775303, 0.0074925 , 0.21143104, 0.2018041 , 0.0165253 ,
       0.17152404, 0.14891828, 0.18380569, 0.03639995, 0.09860293,
```

Pipelines in sklearn: GridSearch with Pipelines

Pipelines in sklearn: GridSearch with Pipelines

- specify grid points using 'step name' + '__' (double-underscore) + 'argument'

Pipelines in sklearn: GridSearch with Pipelines

- specify grid points using 'step name' + '__' (double-underscore) + 'argument'

In [6]:
```
from sklearn.model_selection import GridSearchCV

# separate step-names and argument-names with double-underscore '__'
params = {'pca__n_components':[2,10,20],
          'lr__penalty':['none','l1','l2'],
          'lr__C':[.01,1,10,100]}

gscv = GridSearchCV(pipe1, params, cv=3, n_jobs=-1).fit(X_bc_train,y_bc_train)

gscv.best_params_
```

Out[6]:
```
{'lr__C': 1, 'lr__penalty': 'l1', 'pca__n_components': 20}
```

Pipelines in sklearn: GridSearch with Pipelines

- specify grid points using 'step name' + '__' (double-underscore) + 'argument'

In [6]:
```
from sklearn.model_selection import GridSearchCV

# separate step-names and argument-names with double-underscore '__'
params = {'pca__n_components':[2,10,20],
          'lr__penalty':['none','l1','l2'],
          'lr__C':[.01,1,10,100]}

gscv = GridSearchCV(pipe1, params, cv=3, n_jobs=-1).fit(X_bc_train,y_bc_train)

gscv.best_params_
```

Out[6]:
```
{'lr__C': 1, 'lr__penalty': 'l1', 'pca__n_components': 20}
```

In [7]:
```
score = gscv.score(X_bc_test,y_bc_test)
print(f'test set accuracy: {score:0.3f}')
```

```
test set accuracy: 0.965
```

Pipelines in sklearn: GridSearch with Pipelines

- specify grid points using 'step name' + '__' (double-underscore) + 'argument'

In [6]:
```
from sklearn.model_selection import GridSearchCV

# separate step-names and argument-names with double-underscore '__'
params = {'pca__n_components':[2,10,20],
          'lr__penalty':['none','l1','l2'],
          'lr__C':[.01,1,10,100]}

gscv = GridSearchCV(pipe1, params, cv=3, n_jobs=-1).fit(X_bc_train,y_bc_train)

gscv.best_params_
```

Out[6]:
```
{'lr__C': 1, 'lr__penalty': 'l1', 'pca__n_components': 20}
```

In [7]:
```
score = gscv.score(X_bc_test,y_bc_test)
print(f'test set accuracy: {score:0.3f}')
```

```
test set accuracy: 0.965
```

In [8]:
```
gscv.best_estimator_
```

Out[8]:
```
Pipeline(steps=[('scale', StandardScaler()), ('pca', PCA(n_components=20)),
                ('lr',
                 LogisticRegression(C=1, max_iter=1000, penalty='l1',
                                    random_state=123, solver='saga'))])
```

Pipelines in sklearn with `make_pipeline`

- shorthand for Pipeline
- step names are lowercase of class names

Pipelines in sklearn with `make_pipeline`

- shorthand for Pipeline
- step names are lowercase of class names

In [9]:
```
from sklearn.pipeline import make_pipeline

# make_pipeline: arguments in order of how they should be applied
pipe2 = make_pipeline(StandardScaler(),                   # center and scale data
                     PCA(n_components=2),                 # extract 2 dimensions
                     LogisticRegression(random_state=123) # classify using logistic regression
                    )
pipe2.fit(X_bc_train,y_bc_train)

pipe2
```

Out[9]:
```
Pipeline(steps=[('standardscaler', StandardScaler()),
                ('pca', PCA(n_components=2)),
                ('logisticregression', LogisticRegression(random_state=123))])
```

Pipelines in sklearn with `make_pipeline`

- shorthand for Pipeline
- step names are lowercase of class names

In [9]:
```
from sklearn.pipeline import make_pipeline

# make_pipeline: arguments in order of how they should be applied
pipe2 = make_pipeline(StandardScaler(),                   # center and scale data
                     PCA(n_components=2),                  # extract 2 dimensions
                     LogisticRegression(random_state=123) # classify using logistic regression
                    )
pipe2.fit(X_bc_train,y_bc_train)

pipe2
```

Out[9]:
```
Pipeline(steps=[('standardscaler', StandardScaler()),
                ('pca', PCA(n_components=2)),
                ('logisticregression', LogisticRegression(random_state=123))])
```

In [10]:
```
pipe2['logisticregression'].coef_
```

Out[10]:
```
array([[-2.0068728 ,  1.12126495]])
```

ColumnTransformer

- Transform sets of columns differently as part of a pipeline
- For example: makes it possible to transform categorical and numeric differently

Binary Classification With Mixed Features, Missing Data

Binary Classification With Mixed Features, Missing Data

In [11]:
```python
# from https://scikit-learn.org/stable/auto_examples/compose/plot_column_transformer_mixed_types.html#sphx-glr-auto-examples-com
titanic_url = ('https://raw.githubusercontent.com/amueller/'
               'scipy-2017-sklearn/091d371/notebooks/datasets/titanic3.csv')
df_titanic = pd.read_csv(titanic_url)[['age','fare','embarked','sex','pclass','survived']]
# Numeric Features:
# - age: float.
# - fare: float.
# Categorical Features:
# - embarked: categories encoded as strings {'C', 'S', 'Q'}.
# - sex: categories encoded as strings {'female', 'male'}.
# - pclass: ordinal integers {1, 2, 3}.
df_titanic.head(1)
```

Out[11]:

	age	fare	embarked	sex	pclass	survived
0	29.0	211.3375	S	female	1	1

Binary Classification With Mixed Features, Missing Data

In [11]:

```
# from https://scikit-learn.org/stable/auto_examples/compose/plot_column_transformer_mixed_types.html#sphx-glr-auto-examples-com
titanic_url = ('https://raw.githubusercontent.com/amueller/'
               'scipy-2017-sklearn/091d371/notebooks/datasets/titanic3.csv')
df_titanic = pd.read_csv(titanic_url)[['age','fare','embarked','sex','pclass','survived']]
# Numeric Features:
# - age: float.
# - fare: float.
# Categorical Features:
# - embarked: categories encoded as strings {'C', 'S', 'Q'}.
# - sex: categories encoded as strings {'female', 'male'}.
# - pclass: ordinal integers {1, 2, 3}.
df_titanic.head(1)
```

Out[11]:

	age	fare	embarked	sex	pclass	survived
0	29.0	211.3375	S	female	1	1

In [12]:

```
df_titanic.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1309 entries, 0 to 1308
Data columns (total 6 columns):
 #   Column    Non-Null Count  Dtype  
---  ------    --------------  -----  
 0   age       1046 non-null   float64
 1   fare      1308 non-null   float64
 2   embarked  1307 non-null   object 
 3   sex       1309 non-null   object 
 4   pclass    1309 non-null   int64  
 5   survived  1309 non-null   int64  
dtypes: float64(2), int64(2), object(2)
memory usage: 61.5+ KB
```

ColumnTransformer Cont.

ColumnTransformer Cont.

In [13]:
```
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder

# specify columns subset
numeric_features = ['age', 'fare']
# specify pipeline to apply to those columns
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')), # fill missing values with median
    ('scaler', StandardScaler())])                 # scale features
```

ColumnTransformer Cont.

In [13]:
```
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder

# specify columns subset
numeric_features = ['age', 'fare']
# specify pipeline to apply to those columns
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')), # fill missing values with median
    ('scaler', StandardScaler())])                 # scale features
```

In [14]:
```
categorical_features = ['embarked', 'sex', 'pclass']
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant', fill_value='missing')), # fill missing value with 'missing'
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])                   # one hot encode
```

ColumnTransformer Cont.

In [13]:
```
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder

# specify columns subset
numeric_features = ['age', 'fare']
# specify pipeline to apply to those columns
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')), # fill missing values with median
    ('scaler', StandardScaler())])                 # scale features
```

In [14]:
```
categorical_features = ['embarked', 'sex', 'pclass']
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant', fill_value='missing')), # fill missing value with 'missing'
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])                   # one hot encode
```

In [15]:
```
# combine column pipelines
preprocessor = ColumnTransformer(
    transformers=[('num', numeric_transformer, numeric_features),
                  ('cat', categorical_transformer, categorical_features)
                 ])
```

ColumnTransformer Cont.

In [13]:
```
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder

# specify columns subset
numeric_features = ['age', 'fare']
# specify pipeline to apply to those columns
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')), # fill missing values with median
    ('scaler', StandardScaler())])                 # scale features
```

In [14]:
```
categorical_features = ['embarked', 'sex', 'pclass']
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant', fill_value='missing')), # fill missing value with 'missing'
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])                   # one hot encode
```

In [15]:
```
# combine column pipelines
preprocessor = ColumnTransformer(
    transformers=[('num', numeric_transformer, numeric_features),
                  ('cat', categorical_transformer, categorical_features)
                 ])
```

In [16]:
```
# add a final prediction step
pipe3 = Pipeline(steps=[('preprocessor', preprocessor),
                        ('classifier', LogisticRegression(solver='lbfgs', random_state=42))
                       ])
```

ColumnTransformer Cont.

ColumnTransformer Cont.

In [17]:
```
X_titanic = df_titanic.drop('survived', axis=1)
y_titanic = df_titanic['survived']

X_titanic_train, X_titanic_test, y_titanic_train, y_titanic_test = train_test_split(X_titanic,
                                                                                     y_titanic,
                                                                                     test_size=0.2,
                                                                                     random_state=42)
pipe3.fit(X_titanic_train, y_titanic_train)
print(f"train set score: {pipe3.score(X_titanic_train, y_titanic_train):.3f}")
print(f"test set score : {pipe3.score(X_titanic_test, y_titanic_test):.3f}")
```

```
train set score: 0.784
test set score : 0.771
```

ColumnTransformer Cont.

In [17]:
```
X_titanic = df_titanic.drop('survived', axis=1)
y_titanic = df_titanic['survived']

X_titanic_train, X_titanic_test, y_titanic_train, y_titanic_test = train_test_split(X_titanic,
                                                                                    y_titanic,
                                                                                    test_size=0.2,
                                                                                    random_state=42)

pipe3.fit(X_titanic_train, y_titanic_train)
print(f"train set score: {pipe3.score(X_titanic_train, y_titanic_train):.3f}")
print(f"test set score : {pipe3.score(X_titanic_test, y_titanic_test):.3f}")
```

```
train set score: 0.784
test set score : 0.771
```

In [18]:
```
from sklearn.model_selection import GridSearchCV

# grid search deep inside the pipeline
param_grid = {
    'preprocessor__num__imputer__strategy': ['mean', 'median'],
    'classifier__C': [0.1, 1.0, 10, 100],
}

gs_pipeline = GridSearchCV(pipe3, param_grid, cv=3)
gs_pipeline.fit(X_titanic_train, y_titanic_train)
print("best test set score from grid search: {:.3f}".format(gs_pipeline.score(X_titanic_test, y_titanic_test)))
print("best parameter settings: {}".format(gs_pipeline.best_params_))
```

```
best test set score from grid search: 0.771
best parameter settings: {'classifier__C': 100, 'preprocessor__num__imputer__strategy': 'median'}
```

Questions re Pipelines?

Natural Language Processing (NLP)

- Analyzing and interacting with natural language
- Python Libraries
  - **sklearn**
  - nltk
  - **spaCy**
  - gensim
  - ...

Natural Language Processing (NLP)

- Many NLP Tasks
  - **sentiment analysis**
  - **topic modeling**
  - entity detection
  - machine translation
  - natural language generation
  - question answering
  - relationship extraction
  - automatic summarization
  - ...

Recall: Python Builtin String Functions

Recall: Python Builtin String Functions

In [19]:
```
doc = "D.S. is fun!"
doc
```

Out[19]:
```
'D.S. is fun!'
```

Recall: Python Builtin String Functions

In [19]:
```
doc = "D.S. is fun!"
doc
```

Out[19]:
```
'D.S. is fun!'
```

In [20]:
```
doc.lower(),doc.upper()       # change capitalization
```

Out[20]:
```
('d.s. is fun!', 'D.S. IS FUN!')
```

Recall: Python Builtin String Functions

In [19]:
```
doc = "D.S. is fun!"
doc
```

Out[19]:
```
'D.S. is fun!'
```

In [20]:
```
doc.lower(),doc.upper()       # change capitalization
```

Out[20]:
```
('d.s. is fun!', 'D.S. IS FUN!')
```

In [21]:
```
doc.split() , doc.split('.')  # split a string into parts (default is whitespace)
```

Out[21]:
```
(['D.S.', 'is', 'fun!'], ['D', 'S', ' is fun!'])
```

Recall: Python Builtin String Functions

```
In [19]: doc = "D.S. is fun!"
         doc

Out[19]: 'D.S. is fun!'

In [20]: doc.lower(),doc.upper()       # change capitalization

Out[20]: ('d.s. is fun!', 'D.S. IS FUN!')

In [21]: doc.split() , doc.split('.')  # split a string into parts (default is whitespace)

Out[21]: (['D.S.', 'is', 'fun!'], ['D', 'S', ' is fun!'])

In [22]: '|'.join(['ab','c','d'])      # join items in a list together

Out[22]: 'ab|c|d'
```

Recall: Python Builtin String Functions

In [19]:
```
doc = "D.S. is fun!"
doc
```

Out[19]:
```
'D.S. is fun!'
```

In [20]:
```
doc.lower(),doc.upper()       # change capitalization
```

Out[20]:
```
('d.s. is fun!', 'D.S. IS FUN!')
```

In [21]:
```
doc.split() , doc.split('.')  # split a string into parts (default is whitespace)
```

Out[21]:
```
(['D.S.', 'is', 'fun!'], ['D', 'S', ' is fun!'])
```

In [22]:
```
'|'.join(['ab','c','d'])      # join items in a list together
```

Out[22]:
```
'ab|c|d'
```

In [23]:
```
'|'.join(doc[:5])             # a string itself is treated like a list of characters
```

Out[23]:
```
'D|.|S|.| '
```

Recall: Python Builtin String Functions

```
In [19]: doc = "D.S. is fun!"
         doc

Out[19]: 'D.S. is fun!'

In [20]: doc.lower(),doc.upper()       # change capitalization

Out[20]: ('d.s. is fun!', 'D.S. IS FUN!')

In [21]: doc.split() , doc.split('.')  # split a string into parts (default is whitespace)

Out[21]: (['D.S.', 'is', 'fun!'], ['D', 'S', ' is fun!'])

In [22]: '|'.join(['ab','c','d'])      # join items in a list together

Out[22]: 'ab|c|d'

In [23]: '|'.join(doc[:5])             # a string itself is treated like a list of characters

Out[23]: 'D|.|S|.| '

In [24]: '  test   '.strip()           # remove whitespace from the beginning and end of a string

Out[24]: 'test'
```

Recall: Python Builtin String Functions

In [19]:
```
doc = "D.S. is fun!"
doc
```

Out[19]:
```
'D.S. is fun!'
```

In [20]:
```
doc.lower(),doc.upper()       # change capitalization
```

Out[20]:
```
('d.s. is fun!', 'D.S. IS FUN!')
```

In [21]:
```
doc.split() , doc.split('.')  # split a string into parts (default is whitespace)
```

Out[21]:
```
(['D.S.', 'is', 'fun!'], ['D', 'S', ' is fun!'])
```

In [22]:
```
'|'.join(['ab','c','d'])      # join items in a list together
```

Out[22]:
```
'ab|c|d'
```

In [23]:
```
'|'.join(doc[:5])             # a string itself is treated like a list of characters
```

Out[23]:
```
'D|.|S|.| '
```

In [24]:
```
'  test   '.strip()           # remove whitespace from the beginning and end of a string
```

Out[24]:
```
'test'
```

- and many more, see https://docs.python.org/3.8/library/string.html

NLP: The Corpus

- **corpus:** collection of documents
    - books
    - articles
    - reviews
    - tweets
    - resumes
    - sentences?
    - ...

NLP: Doc Representation

- Documents usually represented as strings
    - string: a sequence (list) of unicode characters

NLP: Doc Representation

- Documents usually represented as strings
    - string: a sequence (list) of unicode characters

```
In [25]: doc = "D.S. is fun!\nIt's  true."
         print(doc)

D.S. is fun!
It's  true.
```

NLP: Doc Representation

- Documents usually represented as strings
  - string: a sequence (list) of unicode characters

```
In [25]: doc = "D.S. is fun!\nIt's  true."
         print(doc)

D.S. is fun!
It's  true.
```

```
In [26]: '|'.join(doc)

Out[26]: "D|.|S|.| |i|s| |f|u|n|!|\n|I|t|'|s| | |t|r|u|e|."
```

NLP: Doc Representation

- Documents usually represented as strings
    - string: a sequence (list) of unicode characters

```
In [25]: doc = "D.S. is fun!\nIt's  true."
         print(doc)

         D.S. is fun!
         It's  true.
```

```
In [26]: '|'.join(doc)

Out[26]: "D|.|S|.| |i|s| |f|u|n|!|\n|I|t|'|s| | |t|r|u|e|."
```

- Need to split this up into parts (**tokens**)
- Good job for **Regular Expressions**

Aside: Regular Expressions

- Strings that define search patterns over text
- Useful for finding/replacing/grouping
- python `re` library (others available)

Aside: Regular Expressions

- Strings that define search patterns over text
- Useful for finding/replacing/grouping
- python `re` library (others available)

In [27]:
```
print(doc)
```

```
D.S. is fun!
It's  true.
```

Aside: Regular Expressions

- Strings that define search patterns over text
- Useful for finding/replacing/grouping
- python `re` library (others available)

In [27]:
```
print(doc)
```

```
D.S. is fun!
It's  true.
```

In [28]:
```
import re
# Find all of the whitespaces in doc
# '\s+' means "one or more whitespace characters"
re.findall(r'\s+',doc)
```

Out[28]:
```
[' ', ' ', '\n', '  ']
```

Aside: Regular Expressions

Just some of the special character definitions:

- `.` : any single character except newline (r'.' matches 'x')
- `*` : match 0 or more repetitions (r'x*' matches 'x','xx','')
- `+` : match 1 or more repetitions (r'x+' matches 'x','xx')
- `?` : match 0 or 1 repetitions (r'x?' matches 'x' or '')

- `^` : beginning of string (r'^D' matches 'D.S.')
- `$` : end of string (r'fun!$' matches 'DS is fun!'`)

Aside: Regular Expression Cont.

- `[]` : a set of characters (^ as first element = not)
- `\s` : whitespace character (Ex: [\t\n\r\f\v])
- `\S` : non-whitespace character (Ex: [^ \t\n\r\f\v])
- `\w` : word character (Ex: [a-zA-Z0-9_])
- `\W` : non-word character
- `\b` : boundary between \w and \W
- and many more!

- See regex101.com for examples and testing

Aside: Regex Python Functions

Aside: Regex Python Functions

In [29]:
```
r'\w*u\w*' # a string of word characters containing u
```

Out[29]:
```
'\\w*u\\w*'
```

Aside: Regex Python Functions

In [29]:
```
r'\w*u\w*' # a string of word characters containing u
```

Out[29]:
```
'\\w*u\\w*'
```

In [30]:
```
re.findall(r'\w*u\w*',doc) # return all substrings that match a pattern
```

Out[30]:
```
['fun', 'true']
```

Aside: Regex Python Functions

In [29]:
```
r'\w*u\w*' # a string of word characters containing u
```
Out[29]:
```
'\\w*u\\w*'
```

In [30]:
```
re.findall(r'\w*u\w*',doc) # return all substrings that match a pattern
```
Out[30]:
```
['fun', 'true']
```

In [31]:
```
re.sub(r'\w*u\w*','XXXX',doc) # substitute all substrings that match a pattern
```
Out[31]:
```
"D.S. is XXXX!\nIt's  XXXX."
```

Aside: Regex Python Functions

In [29]:
```
r'\w*u\w*' # a string of word characters containing u
```
Out[29]:
```
'\\w*u\\w*'
```

In [30]:
```
re.findall(r'\w*u\w*',doc) # return all substrings that match a pattern
```
Out[30]:
```
['fun', 'true']
```

In [31]:
```
re.sub(r'\w*u\w*','XXXX',doc) # substitute all substrings that match a pattern
```
Out[31]:
```
"D.S. is XXXX!\nIt's  XXXX."
```

In [32]:
```
re.split(r'\w*u\w*',doc) # split substrings on a pattern
```
Out[32]:
```
['D.S. is ', "!\nIt's  ", '.']
```

NLP: Tokenization

- **tokens:** strings that make up a document ('the', 'cat',...)
- **tokenization:** convert a document into tokens
- **vocabulary:** set of unique tokens (terms) in corpus

NLP: Tokenization

- **tokens:** strings that make up a document ('the', 'cat',...)
- **tokenization:** convert a document into tokens
- **vocabulary:** set of unique tokens (terms) in corpus

In [33]:
```
# split on whitespace
re.split(r'\s+', doc)
```

Out[33]:
```
['D.S.', 'is', 'fun!', "It's", 'true.']
```

NLP: Tokenization

- **tokens:** strings that make up a document ('the', 'cat',...)
- **tokenization:** convert a document into tokens
- **vocabulary:** set of unique tokens (terms) in corpus

In [33]:
```python
# split on whitespace
re.split(r'\s+', doc)
```

Out[33]:
```
['D.S.', 'is', 'fun!', "It's", 'true.']
```

In [34]:
```python
# find tokens of length 2+ word characters
re.findall(r'\b\w\w+\b',doc)
```

Out[34]:
```
['is', 'fun', 'It', 'true']
```

NLP: Tokenization

- **tokens:** strings that make up a document ('the', 'cat',...)
- **tokenization:** convert a document into tokens
- **vocabulary:** set of unique tokens (terms) in corpus

In [33]:
```
# split on whitespace
re.split(r'\s+', doc)
```

Out[33]: `['D.S.', 'is', 'fun!', "It's", 'true.']`

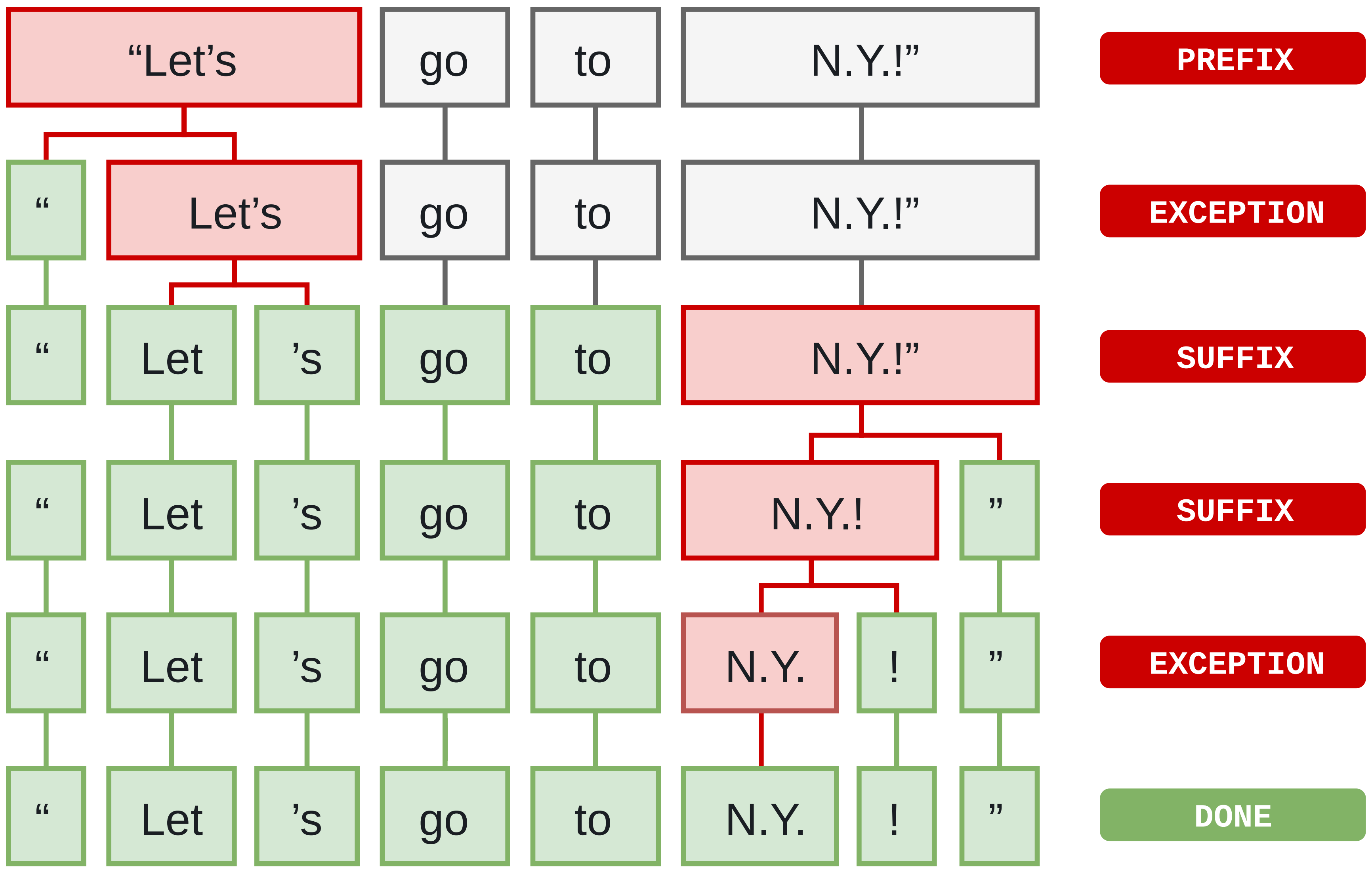
In [34]:
```
# find tokens of length 2+ word characters
re.findall(r'\b\w\w+\b',doc)
```

Out[34]: `['is', 'fun', 'It', 'true']`

In [35]:
```
# find tokens of length 2+ non-space characters
re.findall(r"\b\S\S+\b", doc)
```

Out[35]: `['D.S', 'is', 'fun', "It's", 'true']`

NLP:Tokenization



From https://spacy.io/usage/linguistic-features

NLP: Other Preprocessing

- lowercase
- remove special characters
- add `<START>`, `<END>` tags
- stemming: cut off beginning or ending of word
    - 'studies' becomes 'studi'
    - 'studying' becomes 'study'
- lemmatization: perform morphological analysis
    - 'studies' becomes 'study'
    - 'studying' becomes 'study'

NLP: Bag of Words

- BOW representation: ignore token order

NLP: Bag of Words

- BOW representation: ignore token order

In [36]:
```python
sorted(re.findall(r'\b\S\S+\b', doc.lower()))
```

Out[36]:
```
['d.s', 'fun', 'is', "it's", 'true']
```

NLP: n-Grams

- Unigram: single token
- Bigram: combination of two ordered tokens
- n-Gram: combination of n ordered tokens
- The larger n is, the larger the vocabulary

NLP: n-Grams

- Unigram: single token
- Bigram: combination of two ordered tokens
- n-Gram: combination of n ordered tokens
- The larger n is, the larger the vocabulary

In [37]:

```python
# Bigram example:
tokens = '<start> data science is fun <end>'.split()
[tokens[i]+'_'+tokens[i+1] for i in range(len(tokens)-1)]
```

Out[37]:

```
['<start>_data', 'data_science', 'science_is', 'is_fun', 'fun_<end>']
```

NLP: TF and DF

- **Term Frequency:** number of times a term is seen per document
- $\text{tf}(t, d) = \text{count of term } t \text{ in document } d$

NLP: TF and DF

- **Term Frequency:** number of times a term is seen per document
- $\mathrm{tf}(t, d) = \text{count of term } t \text{ in document } d$

In [38]:
```
corpus = ['red green blue', 'red blue blue']

#Vocabulary
vocab = sorted(set(' '.join(corpus).split()))
vocab
```

Out[38]:
```
['blue', 'green', 'red']
```

NLP: TF and DF

- **Term Frequency:** number of times a term is seen per document
- $\mathrm{tf}(t, d) = \text{count of term } t \text{ in document } d$

In [38]:
```
corpus = ['red green blue', 'red blue blue']

#Vocabulary
vocab = sorted(set(' '.join(corpus).split()))
vocab
```

Out[38]: ['blue', 'green', 'red']

In [39]:
```
#TF
from collections import Counter
tf = np.zeros((len(corpus),len(vocab)))
for i,doc in enumerate(corpus):
    for j,term in enumerate(vocab):
        tf[i,j] = Counter(doc.split())[term]
tf = pd.DataFrame(tf,index=['doc1','doc2'],columns=vocab)
tf
```

Out[39]:

	blue	green	red
doc1	1.0	1.0	1.0
doc2	2.0	0.0	1.0

NLP: TF and DF

- **Document Frequency:** number of documents containing each term
  $\mathrm{df}(t) = \text{count of documents containing term } t$

NLP: TF and DF

- **Document Frequency:** number of documents containing each term

  $\mathrm{df}(t) = \text{count of documents containing term } t$

In [40]:
```
#DF
tf.astype(bool).sum(axis=0)
```

Out[40]:
```
blue     2
green    1
red      2
dtype: int64
```

NLP: Stopwords

- terms that have high (or very low) DF and aren't informative
  - common engish terms (ex: 'a', 'the','in',...)
  - domain specific (ex, in class slides: 'data_science')
  - often removed prior to analysis
  - in sklearn
    - `min_df`, an integer > 0, keep terms that occur in at at least n documents
    - `max_df`, a float in (0,1], keep terms that occur in less than f% of total documents

NLP: CountVectorizer in sklearn

NLP: CountVectorizer in sklearn

In [41]:
```
corpus = ['blue green red', 'blue green green']

from sklearn.feature_extraction.text import CountVectorizer
cvect = CountVectorizer(lowercase=True,    # default, transform all docs to lowercase
                        ngram_range=(1,1), # default, only unigrams
                        min_df=1,          # default, keep all terms
                        max_df=1.0,        # default, keep all terms
                       )
X_cv = cvect.fit_transform(corpus)
X_cv.shape
```

Out[41]:
```
(2, 3)
```

NLP: CountVectorizer in sklearn

In [41]:
```
corpus = ['blue green red', 'blue green green']

from sklearn.feature_extraction.text import CountVectorizer
cvect = CountVectorizer(lowercase=True,    # default, transform all docs to lowercase
                        ngram_range=(1,1), # default, only unigrams
                        min_df=1,          # default, keep all terms
                        max_df=1.0,        # default, keep all terms
                       )
X_cv = cvect.fit_transform(corpus)
X_cv.shape
```

Out[41]:
```
(2, 3)
```

In [42]:
```
cvect.vocabulary_ # learned vocabulary, term:index pairs
```

Out[42]:
```
{'blue': 0, 'green': 1, 'red': 2}
```

NLP: CountVectorizer in sklearn

In [41]:
```
corpus = ['blue green red', 'blue green green']

from sklearn.feature_extraction.text import CountVectorizer
cvect = CountVectorizer(lowercase=True,    # default, transform all docs to lowercase
                        ngram_range=(1,1), # default, only unigrams
                        min_df=1,          # default, keep all terms
                        max_df=1.0,        # default, keep all terms
                       )
X_cv = cvect.fit_transform(corpus)
X_cv.shape
```

Out[41]:
```
(2, 3)
```

In [42]:
```
cvect.vocabulary_ # learned vocabulary, term:index pairs
```

Out[42]:
```
{'blue': 0, 'green': 1, 'red': 2}
```

In [43]:
```
cvect.get_feature_names() # vocabulary, sorted by indexs
```

Out[43]:
```
['blue', 'green', 'red']
```

NLP: CountVectorizer in sklearn

In [41]:
```
corpus = ['blue green red', 'blue green green']

from sklearn.feature_extraction.text import CountVectorizer
cvect = CountVectorizer(lowercase=True,    # default, transform all docs to lowercase
                        ngram_range=(1,1), # default, only unigrams
                        min_df=1,          # default, keep all terms
                        max_df=1.0,        # default, keep all terms
                       )
X_cv = cvect.fit_transform(corpus)
X_cv.shape
```

Out[41]:
```
(2, 3)
```

In [42]:
```
cvect.vocabulary_ # learned vocabulary, term:index pairs
```

Out[42]:
```
{'blue': 0, 'green': 1, 'red': 2}
```

In [43]:
```
cvect.get_feature_names() # vocabulary, sorted by indexs
```

Out[43]:
```
['blue', 'green', 'red']
```

In [44]:
```
X_cv.todense() # term frequencies
```

Out[44]:
```
matrix([[1, 1, 1],
        [1, 2, 0]])
```

NLP: CountVectorizer in sklearn

In [41]:
```
corpus = ['blue green red', 'blue green green']

from sklearn.feature_extraction.text import CountVectorizer
cvect = CountVectorizer(lowercase=True,    # default, transform all docs to lowercase
                        ngram_range=(1,1), # default, only unigrams
                        min_df=1,          # default, keep all terms
                        max_df=1.0,        # default, keep all terms
                       )
X_cv = cvect.fit_transform(corpus)
X_cv.shape
```

Out[41]:
```
(2, 3)
```

In [42]:
```
cvect.vocabulary_ # learned vocabulary, term:index pairs
```

Out[42]:
```
{'blue': 0, 'green': 1, 'red': 2}
```

In [43]:
```
cvect.get_feature_names() # vocabulary, sorted by indexs
```

Out[43]:
```
['blue', 'green', 'red']
```

In [44]:
```
X_cv.todense() # term frequencies
```

Out[44]:
```
matrix([[1, 1, 1],
        [1, 2, 0]])
```

In [45]:
```
cvect.inverse_transform(X_cv) # mapping back to terms via vocabulary mapping
```

Out[45]:
```
[array(['blue', 'green', 'red'], dtype='<U5'),
 array(['blue', 'green'], dtype='<U5')]
```

NLP: TfIdf

- What if some terms are still uninformative?
- Can we downweight terms that occur in many documents?
- **Term Frequency * Inverse Document Frequency (tf-idf)**
  - $\text{tf-idf}(t, d) = \text{tf}(t, d) \times \text{idf}(t)$
  - $\text{idf}(t) = \log \frac{1+n}{1+\text{df}(t)} + 1$

NLP: TfIdf

- What if some terms are still uninformative?
- Can we downweight terms that occur in many documents?
- **Term Frequency * Inverse Document Frequency (tf-idf)**
  - $\text{tf-idf}(t, d) = \text{tf}(t, d) \times \text{idf}(t)$
  - $\text{idf}(t) = \log \frac{1+n}{1+\text{df}(t)} + 1$

In [46]:
```python
from sklearn.feature_extraction.text import TfidfVectorizer

tfidfvect = TfidfVectorizer(norm='l2') # by default, also doing l2 normalization

X_tfidf = tfidfvect.fit_transform(corpus)
sorted(tfidfvect.vocabulary_.items(),key=lambda x: x[1])
```

Out[46]:
```
[('blue', 0), ('green', 1), ('red', 2)]
```

NLP: TfIdf

- What if some terms are still uninformative?
- Can we downweight terms that occur in many documents?
- **Term Frequency * Inverse Document Frequency (tf-idf)**
    - $\text{tf-idf}(t, d) = \text{tf}(t, d) \times \text{idf}(t)$
    - $\text{idf}(t) = \log \frac{1+n}{1+\text{df}(t)} + 1$

In [46]:
```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidfvect = TfidfVectorizer(norm='l2') # by default, also doing l2 normalization

X_tfidf = tfidfvect.fit_transform(corpus)
sorted(tfidfvect.vocabulary_.items(),key=lambda x: x[1])
```

Out[46]:
```
[('blue', 0), ('green', 1), ('red', 2)]
```

In [47]:
```
X_tfidf.todense()
```

Out[47]:
```
matrix([[0.50154891, 0.50154891, 0.70490949],
        [0.4472136 , 0.89442719, 0.        ]])
```

NLP: TfIdf

- What if some terms are still uninformative?
- Can we downweight terms that occur in many documents?
- **Term Frequency * Inverse Document Frequency (tf-idf)**
  - $\text{tf-idf}(t, d) = \text{tf}(t, d) \times \text{idf}(t)$
  - $\text{idf}(t) = \log \frac{1+n}{1+\text{df}(t)} + 1$

In [46]:

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidfvect = TfidfVectorizer(norm='l2') # by default, also doing l2 normalization

X_tfidf = tfidfvect.fit_transform(corpus)
sorted(tfidfvect.vocabulary_.items(),key=lambda x: x[1])
```

Out[46]:

```
[('blue', 0), ('green', 1), ('red', 2)]
```

In [47]:

```
X_tfidf.todense()
```

Out[47]:

```
matrix([[0.50154891, 0.50154891, 0.70490949],
        [0.4472136 , 0.89442719, 0.        ]])
```

In [48]:

```
# can also use to get term frequencies by setting use_idf to False and norm to none
TfidfVectorizer(use_idf=False, norm=None).fit_transform(corpus).todense()
```

Out[48]:

```
matrix([[1., 1., 1.],
        [1., 2., 0.]])
```

NLP: Classification Example

NLP: Classification Example

In [49]:
```
from sklearn.datasets import fetch_20newsgroups

ngs = fetch_20newsgroups(categories=['rec.sport.baseball','rec.sport.hockey']) # dataset has 20 categories, only get two

docs_ngs = ngs['data']                              # get documents (emails)
y_ngs = ngs['target']                               # get targets ([0,1])
target_names_ngs = ngs['target_names']              # get target names (['rec.sport.baseball','rec.sport.hockey'])

print(y_ngs[0], target_names_ngs[y_ngs[0]])  # print target int and target name
print('-'*50)                                       # print a string of 50 dashes
print(docs_ngs[0].strip()[:600])                    # print beginning characters of first doc, after stripping whitespace
```

```
0 rec.sport.baseball
--------------------------------------------------
From: dougb@comm.mot.com (Doug Bank)
Subject: Re: Info needed for Cleveland tickets
Reply-To: dougb@ecs.comm.mot.com
Organization: Motorola Land Mobile Products Sector
Distribution: usa
Nntp-Posting-Host: 145.1.146.35
Lines: 17

In article <1993Apr1.234031.4950@leland.Stanford.EDU>, bohnert@leland.Stanford.EDU (matthew bohnert) writes:

|> I'm going to be in Cleveland Thursday, April 15 to Sunday, April 18.
|> Does anybody know if the Tribe will be in town on those dates, and
|> if so, who're they playing and if tickets are available?

The tribe will be in town from April 16 to the 19th.
There
```

NLP Example: Transform Docs

NLP Example: Transform Docs

In [50]:

```python
from sklearn.model_selection import train_test_split
docs_ngs_train,docs_ngs_test,y_ngs_train,y_ngs_test = train_test_split(docs_ngs,y_ngs)

vect = TfidfVectorizer(lowercase=True,
                       min_df=5,       # occur in at least 5 documents
                       max_df=0.8,     # occur in at most 80% of documents
                       token_pattern='\\b\\S\\S+\\b',  # tokens of at least 2 non-space characters
                       ngram_range=(1,1),  # only unigrams
                       use_idf=False,  # term frequency counts instead of tf-idf
                       norm=None       # do not normalize
                      )
X_ngs_train = vect.fit_transform(docs_ngs_train)
X_ngs_train.shape
```

Out[50]: (897, 3760)

NLP Example: Transform Docs

In [50]:
```
from sklearn.model_selection import train_test_split
docs_ngs_train,docs_ngs_test,y_ngs_train,y_ngs_test = train_test_split(docs_ngs,y_ngs)

vect = TfidfVectorizer(lowercase=True,
                       min_df=5,       # occur in at least 5 documents
                       max_df=0.8,     # occur in at most 80% of documents
                       token_pattern='\\b\\S\\S+\\b',  # tokens of at least 2 non-space characters
                       ngram_range=(1,1),  # only unigrams
                       use_idf=False,  # term frequency counts instead of tf-idf
                       norm=None       # do not normalize
                      )
X_ngs_train = vect.fit_transform(docs_ngs_train)
X_ngs_train.shape
```

Out[50]:
```
(897, 3760)
```

In [51]:
```
# first few terms in learned vocabulary
list(vect.vocabulary_.items())[:5]
```

Out[51]:
```
[('king', 1913),
 ('re', 2743),
 ('players', 2576),
 ('40', 176),
 ('college', 882)]
```

NLP Example: Transform Docs

In [50]:
```python
from sklearn.model_selection import train_test_split
docs_ngs_train,docs_ngs_test,y_ngs_train,y_ngs_test = train_test_split(docs_ngs,y_ngs)

vect = TfidfVectorizer(lowercase=True,
                       min_df=5,        # occur in at least 5 documents
                       max_df=0.8,      # occur in at most 80% of documents
                       token_pattern='\\b\\S\\S+\\b',  # tokens of at least 2 non-space characters
                       ngram_range=(1,1),  # only unigrams
                       use_idf=False,   # term frequency counts instead of tf-idf
                       norm=None        # do not normalize
                      )
X_ngs_train = vect.fit_transform(docs_ngs_train)
X_ngs_train.shape
```

Out[50]:
```
(897, 3760)
```

In [51]:
```python
# first few terms in learned vocabulary
list(vect.vocabulary_.items())[:5]
```

Out[51]:
```
[('king', 1913),
 ('re', 2743),
 ('players', 2576),
 ('40', 176),
 ('college', 882)]
```

In [52]:
```python
# first few terms in learned stopword list
list(vect.stop_words_)[:5]
```

Out[52]:
```
['design', 'saberhagen', '_americans_', 'shayne', 'coons']
```

NLP Example: Train and Evaluate Classifier

NLP Example: Train and Evaluate Classifier

In [54]:
```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.dummy import DummyClassifier

scores_dummy = cross_val_score(DummyClassifier(strategy='most_frequent'),X_ngs_train,y_ngs_train)
scores_lr = cross_val_score(LogisticRegression(),X_ngs_train,y_ngs_train)

print(f'dummy cv accuracy: {scores_dummy.mean():0.2f} +- {scores_dummy.std():0.2f}')
print(f'lr    cv accuracy: {scores_lr.mean():0.2f} +- {scores_lr.std():0.2f}')
```

```
dummy cv accuracy: 0.51 +- 0.00
lr    cv accuracy: 0.95 +- 0.01
```

NLP Example: Using Pipeline

NLP Example: Using Pipeline

In [55]:
```
from sklearn.pipeline import Pipeline

# use Pipeline instead of make_pipeline to add names to the steps
# (name,object) tuple pairs for each step
pipe_ngs = Pipeline([('vect',TfidfVectorizer(lowercase=True,
                                             min_df=5,
                                             max_df=0.8,
                                             token_pattern='\\b\\S\\S+\\b',
                                             ngram_range=(1,1),
                                             use_idf=False,
                                             norm=None )
                     ),
                     ('lr',LogisticRegression())
                    ])

pipe_ngs.fit(docs_ngs_train,y_ngs_train) # pass in docs, not transformed X

score_ngs = pipe_ngs.score(docs_ngs_train,y_ngs_train)
print(f'pipeline accuracy on training set: {score_ngs:0.2f}')
```

```
pipeline accuracy on training set: 1.00
```

NLP Example: Using Pipeline

In [55]:
```
from sklearn.pipeline import Pipeline

# use Pipeline instead of make_pipeline to add names to the steps
# (name,object) tuple pairs for each step
pipe_ngs = Pipeline([('vect',TfidfVectorizer(lowercase=True,
                                             min_df=5,
                                             max_df=0.8,
                                             token_pattern='\\b\\S\\S+\\b',
                                             ngram_range=(1,1),
                                             use_idf=False,
                                             norm=None )
                     ),
                     ('lr',LogisticRegression())
                    ])

pipe_ngs.fit(docs_ngs_train,y_ngs_train) # pass in docs, not transformed X

score_ngs = pipe_ngs.score(docs_ngs_train,y_ngs_train)
print(f'pipeline accuracy on training set: {score_ngs:0.2f}')
```

```
pipeline accuracy on training set: 1.00
```

In [56]:
```
scores_pipe = cross_val_score(pipe_ngs,docs_ngs_train,y_ngs_train)
print(f'pipe cv accuracy: {scores_pipe.mean():0.2f} +- {scores_pipe.std():0.2f}')
```

```
pipe cv accuracy: 0.95 +- 0.02
```

NLP Example: Using Pipeline

In [55]:
```
from sklearn.pipeline import Pipeline

# use Pipeline instead of make_pipeline to add names to the steps
# (name,object) tuple pairs for each step
pipe_ngs = Pipeline([('vect',TfidfVectorizer(lowercase=True,
                                             min_df=5,
                                             max_df=0.8,
                                             token_pattern='\\b\\S\\S+\\b',
                                             ngram_range=(1,1),
                                             use_idf=False,
                                             norm=None )
                     ),
                     ('lr',LogisticRegression())
                    ])

pipe_ngs.fit(docs_ngs_train,y_ngs_train) # pass in docs, not transformed X

score_ngs = pipe_ngs.score(docs_ngs_train,y_ngs_train)
print(f'pipeline accuracy on training set: {score_ngs:0.2f}')
```

```
pipeline accuracy on training set: 1.00
```

In [56]:
```
scores_pipe = cross_val_score(pipe_ngs,docs_ngs_train,y_ngs_train)
print(f'pipe cv accuracy: {scores_pipe.mean():0.2f} +- {scores_pipe.std():0.2f}')
```

```
pipe cv accuracy: 0.95 +- 0.02
```

In [57]:
```
list(pipe_ngs['vect'].vocabulary_.items())[:3]
```

Out[57]:
```
[('king', 1913), ('re', 2743), ('players', 2576)]
```

NLP Example: Add Feature Selection

NLP Example: Add Feature Selection

In [58]:
```
from sklearn.feature_selection import SelectFromModel,SelectPercentile

pipe_ngs = Pipeline([('vect',TfidfVectorizer(lowercase=True,
                                             min_df=5,
                                             max_df=0.8,
                                             token_pattern='\\b\\S\\S+\\b',
                                             ngram_range=(1,1),
                                             use_idf=False,
                                             norm=None )
                     ),
                     ('fs',SelectFromModel(estimator=LogisticRegression(C=1.0,
                                                                         penalty='l1',
                                                                         solver='liblinear',
                                                                         max_iter=1000,
                                                                         random_state=123
                                                                        ))),
                     ('lr',LogisticRegression(max_iter=1000))
                    ])
pipe_ngs.fit(docs_ngs_train,y_ngs_train)
print(f'pipeline accuracy on training set: {pipe_ngs.score(docs_ngs_train,y_ngs_train):0.2f}')
scores_pipe = cross_val_score(pipe_ngs,docs_ngs_train,y_ngs_train)
print(f'pipe cv accuracy: {scores_pipe.mean():0.2f} +- {scores_pipe.std():0.2f}')
```

```
pipeline accuracy on training set: 1.00
pipe cv accuracy: 0.93 +- 0.01
```

NLP Example: Grid Search with Feature Selection

NLP Example: Grid Search with Feature Selection

In [59]:

```
# NOTE: this may take a minute or so
params = {'vect__use_idf':[True,False],
          'vect__ngram_range':[(1,1),(2,2)],
          'fs__estimator__C':[10,1000],
          'lr__C':[.01,1,100]}

gscv = GridSearchCV(pipe_ngs, params, cv=2, n_jobs=-1).fit(docs_ngs_train,y_ngs_train)

print(f'gscsv best parameters  : {gscv.best_params_}')
print(f'gscsv best cv accuracy : {gscv.best_score_:0.2f}')
print(f'gscsv test set accuracy: {gscv.score(docs_ngs_test,y_ngs_test):0.2f}')
```

```
gscsv best parameters  : {'fs__estimator__C': 1000, 'lr__C': 0.01, 'vect__ngram_range': (1, 1), 'vect__use_idf': True}
gscsv best cv accuracy : 0.94
gscsv test set accuracy: 0.97
```

Sentiment Analysis and sklearn

- determine sentiment/opinion from unstructured test
- usually positive/negative, but is domain specific
- can be treated as a classification task (with a target, using all of the tools we know)
- can also be treated as a linguistic task (sentence parsing)

- Example: determine sentiment of movie reviews
- see sentiment_analysis_example.ipynb

Topic Modeling

- What topics are our documents composed of?
- How much of each topic does each document contain?
- Can we represent documents using topic weights? (dimensionality reduction)

Topic Modeling

- What topics are our documents composed of?
- How much of each topic does each document contain?
- Can we represent documents using topic weights? (dimensionality reduction)

- What is topic modeling?
- How does Latent Dirichlet Allocation (LDA) work?
- How to train and use LDA with sklearn?

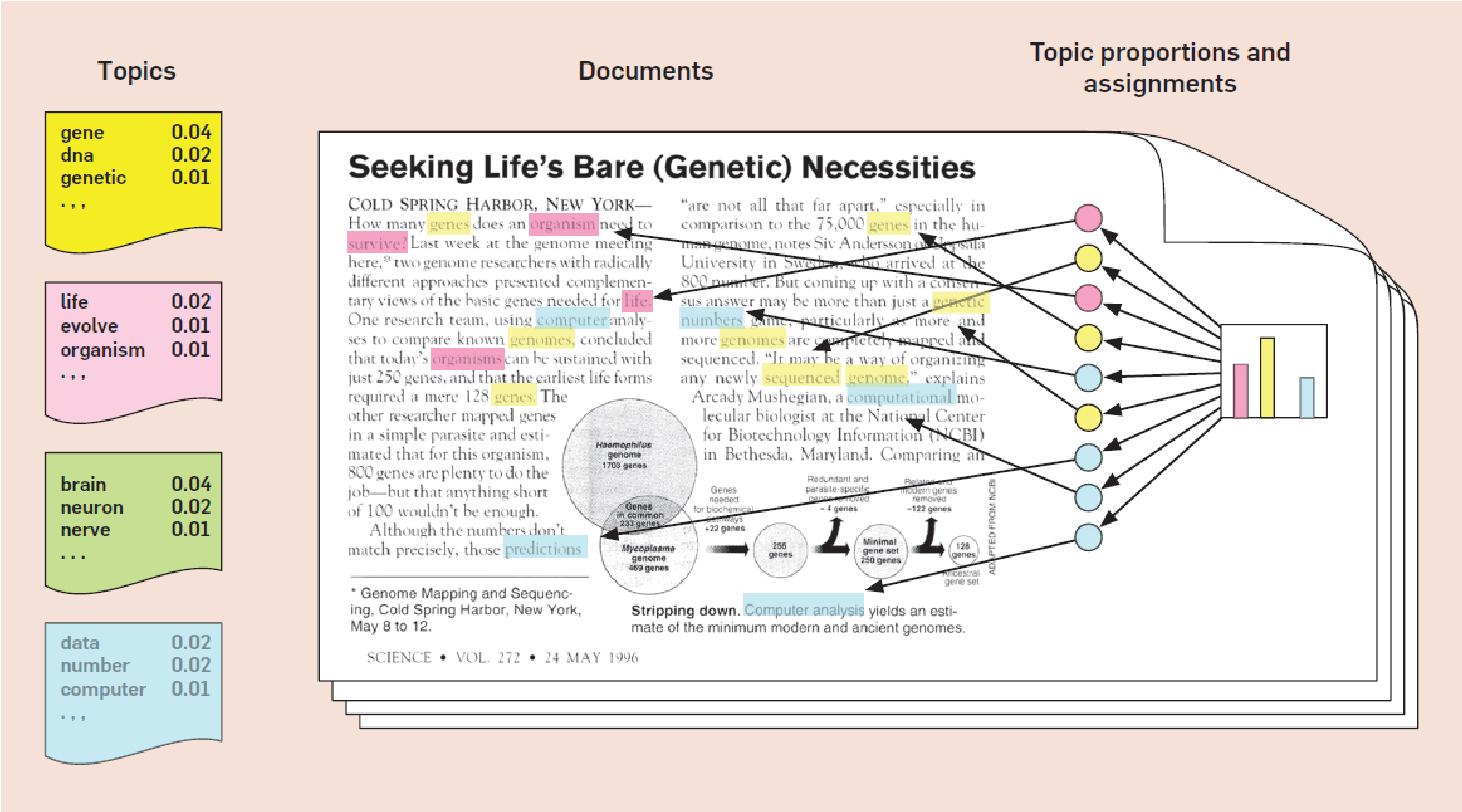
What is Topic Modeling?

- **topic:** a collection of related words
- A document can be composed of several topics

- Given a collection of documents, we can ask:
  - **What terms make up each topic?** (per topic term distribution)
  - **What topics make up each document?** (per document topic distribution)

Topic Modeling with Latent Dirichlet Allocation (LDA)

- Unsupervised method for determining topics and topic assignments



From David Blei

Two Important Matrices Learned by LDA

- the **per topic term distributions** aka φ (phi)

Two Important Matrices Learned by LDA

- the **per topic term distributions** aka φ (phi)

In [60]:
```
topics = ['topic1','topic2']
vocab = ['cat','baseball','play']
phi = pd.DataFrame([[0.4,.2,.4],[0.2,.4,.4]],columns=vocab,index=topics)
phi
```

Out[60]:

	cat	baseball	play
topic1	0.4	0.2	0.4
topic2	0.2	0.4	0.4

Two Important Matrices Learned by LDA

- the **per topic term distributions** aka φ (phi)

In [60]:
```
topics = ['topic1','topic2']
vocab = ['cat','baseball','play']
phi = pd.DataFrame([[0.4,.2,.4],[0.2,.4,.4]],columns=vocab,index=topics)
phi
```

Out[60]:

	cat	baseball	play
topic1	0.4	0.2	0.4
topic2	0.2	0.4	0.4

- the **per document term distributions** aka θ (theta)

Two Important Matrices Learned by LDA

- the **per topic term distributions** aka φ (phi)

In [60]:
```
topics = ['topic1','topic2']
vocab = ['cat','baseball','play']
phi = pd.DataFrame([[0.4,.2,.4],[0.2,.4,.4]],columns=vocab,index=topics)
phi
```

Out[60]:

	cat	baseball	play
topic1	0.4	0.2	0.4
topic2	0.2	0.4	0.4

- the **per document term distributions** aka θ (theta)

In [61]:
```
topics = ['topic1','topic2']
docs = ['doc1','doc2']
theta = pd.DataFrame([[0.1,.9],[.5,.5]],columns=topics,index=docs)
theta
```

Out[61]:

	topic1	topic2
doc1	0.1	0.9
doc2	0.5	0.5

Topic Modeling: Example

- Given the data and the number of topics we want

Topic Modeling: Example

- Given the data and the number of topics we want

```
In [62]: corpus = ['the dog and cat played tennis',
                   'tennis and baseball are sports',
                   'a dog or a cat can be a pet']

         M = 3 # the number of documents

         vocab = ['baseball','cat','dog','pet','played','tennis']

         V = len(vocab) # size of vocabulary

         K = 2 # our guess about the number of topics

         print(f'{M = :}\n{V = :}\n{K = :}')

         M = 3
         V = 6
         K = 2
```

Topic Modeling: Example

- Guessing some **per topic term distributions** (φ) given the documents and vocab

Topic Modeling: Example

- Guessing some **per topic term distributions** (φ) given the documents and vocab

In [63]:
```
print(vocab)
```

```
['baseball', 'cat', 'dog', 'pet', 'played', 'tennis']
```

Topic Modeling: Example

- Guessing some **per topic term distributions** (φ) given the documents and vocab

In [63]:
```
print(vocab)
```

```
['baseball', 'cat', 'dog', 'pet', 'played', 'tennis']
```

In [64]:
```
# the probability of each term given topic 1 (high for sports terms)
topic_1 = [.33,   0,   0,   0, .33, .33]

# the probability of each term given topic 2 (high for pet terms)
topic_2 = [  0, .25, .25, .25, .25,   0]

# per topic term distributions
phi = pd.DataFrame([topic_1, topic_2],columns=vocab,
                   index=['topic_'+str(x) for x in range(1,K+1)])

phi
```

Out[64]:

	baseball	cat	dog	pet	played	tennis
topic_1	0.33	0.00	0.00	0.00	0.33	0.33
topic_2	0.00	0.25	0.25	0.25	0.25	0.00

Topic Modeling: Example

- Guessing the **per document topic distributions** θ given the **topics**

Topic Modeling: Example

- Guessing the **per document topic distributions** θ given the **topics**

In [65]:
```
# Given our guess about phi
display(phi)
# And the corpus
corpus
```

	baseball	cat	dog	pet	played	tennis
topic_1	0.33	0.00	0.00	0.00	0.33	0.33
topic_2	0.00	0.25	0.25	0.25	0.25	0.00

Out[65]:
```
['the dog and cat played tennis',
 'tennis and baseball are sports',
 'a dog or a cat can be a pet']
```

Topic Modeling: Example

- Guessing the **per document topic distributions** θ given the **topics**

In [65]:
```
# Given our guess about phi
display(phi)
# And the corpus
corpus
```

	baseball	cat	dog	pet	played	tennis
topic_1	0.33	0.00	0.00	0.00	0.33	0.33
topic_2	0.00	0.25	0.25	0.25	0.25	0.00

Out[65]:
```
['the dog and cat played tennis',
 'tennis and baseball are sports',
 'a dog or a cat can be a pet']
```

In [66]:
```
# generate a guess about per document topic distributions
theta = pd.DataFrame([[.50, .50],
                      [.99, .01],
                      [.01, .99]],
                     columns=['topic_'+str(x) for x in range(1,K+1)],
                     index=['doc_'+str(x) for x in range(1,M+1)])
theta
```

Out[66]:

	topic_1	topic_2
doc_1	0.50	0.50
doc_2	0.99	0.01
doc_3	0.01	0.99

Topic Modeling With LDA

- Given
  - a set of documents
  - a number of topics K
- Learn
  - the **per topic term distributions** φ **(phi)**, size: $K \times V$
  - the **per document topic distributions** θ **(theta)**, size: $M \times K$
- How to learn φ and θ:
  - Latent Dirichlet Allocation (LDA)
  - generative statistical model
  - Blei, D., Ng, A., Jordan, M. Latent Dirichlet allocation. J. Mach. Learn. Res. 3 (Jan 2003)

Topic Modeling With LDA

- Uses for φ (phi), the per topic term distributions:
  - infering labels for topics
  - word clouds

- Uses for θ (theta), the per document topic distributions:
  - dimentionality reduction
  - clustering
  - similarity

LDA with sklearn

LDA with sklearn

In [67]:
```
# load data from all 20 newsgroups
newsgroups = fetch_20newsgroups()
ngs_all = newsgroups.data
len(ngs_all)
```

Out[67]:
```
11314
```

LDA with sklearn

In [67]:
```
# load data from all 20 newsgroups
newsgroups = fetch_20newsgroups()
ngs_all = newsgroups.data
len(ngs_all)
```

Out[67]: 11314

In [68]:
```
# transform documents using tf-idf
tfidf = TfidfVectorizer(token_pattern=r'\b[a-zA-Z0-9-][a-zA-Z0-9-]+\b',min_df=50, max_df=.2)
X_tfidf = tfidf.fit_transform(ngs_all)
X_tfidf.shape
```

Out[68]: (11314, 4256)

LDA with sklearn

In [67]:
```
# load data from all 20 newsgroups
newsgroups = fetch_20newsgroups()
ngs_all = newsgroups.data
len(ngs_all)
```

Out[67]:
```
11314
```

In [68]:
```
# transform documents using tf-idf
tfidf = TfidfVectorizer(token_pattern=r'\b[a-zA-Z0-9-][a-zA-Z0-9-]+\b',min_df=50, max_df=.2)
X_tfidf = tfidf.fit_transform(ngs_all)
X_tfidf.shape
```

Out[68]:
```
(11314, 4256)
```

In [69]:
```
feature_names = tfidf.get_feature_names()
print(feature_names[:10])
print(feature_names[-10:])
```

```
['00', '000', '01', '02', '03', '04', '05', '06', '07', '08']
['yours', 'yourself', 'ysu', 'zealand', 'zero', 'zeus', 'zip', 'zone', 'zoo', 'zuma']
```

LDA with sklearn Cont.

LDA with sklearn Cont.

In [70]:
```
from sklearn.decomposition import LatentDirichletAllocation

# create model with 20 topics
lda = LatentDirichletAllocation(n_components=20,  # the number of topics
                                n_jobs=-1,        # use all cpus
                                random_state=123) # for reproducability

# learn phi (lda.components_) and theta (X_lda)
# this will take a while!
X_lda = lda.fit_transform(X_tfidf)
```

LDA with sklearn Cont.

In [70]:
```
from sklearn.decomposition import LatentDirichletAllocation

# create model with 20 topics
lda = LatentDirichletAllocation(n_components=20,  # the number of topics
                                n_jobs=-1,        # use all cpus
                                random_state=123) # for reproducability

# learn phi (lda.components_) and theta (X_lda)
# this will take a while!
X_lda = lda.fit_transform(X_tfidf)
```

In [71]:
```
ngs_all[100][:100]
```

Out[71]:
```
'From: tchen@magnus.acs.ohio-state.edu (Tsung-Kun Chen)\nSubject: ** Software forsale (lots) **\nNntp-P'
```

LDA with sklearn Cont.

In [70]:
```
from sklearn.decomposition import LatentDirichletAllocation

# create model with 20 topics
lda = LatentDirichletAllocation(n_components=20,  # the number of topics
                                n_jobs=-1,        # use all cpus
                                random_state=123) # for reproducability

# learn phi (lda.components_) and theta (X_lda)
# this will take a while!
X_lda = lda.fit_transform(X_tfidf)
```

In [71]:
```
ngs_all[100][:100]
```

Out[71]:
```
'From: tchen@magnus.acs.ohio-state.edu (Tsung-Kun Chen)\nSubject: ** Software forsale (lots) **\nNntp-P'
```

In [72]:
```
np.round(X_lda[100],2) # lda representation of document_100
```

Out[72]:
```
array([0.01, 0.01, 0.01, 0.01, 0.1 , 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
       0.01, 0.01, 0.01, 0.38, 0.01, 0.14, 0.01, 0.01, 0.28])
```

LDA with sklearn Cont.

In [70]:
```
from sklearn.decomposition import LatentDirichletAllocation

# create model with 20 topics
lda = LatentDirichletAllocation(n_components=20,  # the number of topics
                                n_jobs=-1,        # use all cpus
                                random_state=123) # for reproducability

# learn phi (lda.components_) and theta (X_lda)
# this will take a while!
X_lda = lda.fit_transform(X_tfidf)
```

In [71]:
```
ngs_all[100][:100]
```

Out[71]:
```
'From: tchen@magnus.acs.ohio-state.edu (Tsung-Kun Chen)\nSubject: ** Software forsale (lots) **\nNntp-P'
```

In [72]:
```
np.round(X_lda[100],2) # lda representation of document_100
```

Out[72]:
```
array([0.01, 0.01, 0.01, 0.01, 0.1 , 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
       0.01, 0.01, 0.01, 0.38, 0.01, 0.14, 0.01, 0.01, 0.28])
```

In [73]:
```
# Note: since this is unsupervised, these numbers may change
np.argsort(X_lda[100])[::-1][:3] # the top topics of document_100
```

Out[73]:
```
array([14, 19, 16])
```

LDA: Per Topic Term Distributions

LDA: Per Topic Term Distributions

In [75]:
```
print_top_words(lda,feature_names,5)
```

```
Topic  0: uga ai georgia covington mcovingt
Topic  1: digex access turkish armenian armenians
Topic  2: god jesus bible christians christian
Topic  3: values objective frank morality ap
Topic  4: ohio-state magnus acs ohio cis
Topic  5: caltech keith sandvik livesey sgi
Topic  6: stratus msg usc indiana sw
Topic  7: alaska uci aurora colostate nsmca
Topic  8: wpi radar psu psuvm detector
Topic  9: columbia utexas gatech cc prism
Topic 10: scsi upenn simms ide bus
Topic 11: nhl team mit players hockey
Topic 12: lehigh duke jewish adobe ns1
Topic 13: henry toronto zoo ti dseg
Topic 14: sale card thanks please mac
Topic 15: virginia joel hall doug douglas
Topic 16: ca his new cs should
Topic 17: cleveland cwru freenet cramer ins
Topic 18: pitt gordon geb banks cs
Topic 19: windows file window files thanks
```

LDA Review

- What did we learn?
    - per document topic distributions
    - per topic term distributions

- What can we use this for?
    - Dimensionality Reduction/Feature Extraction!
    - investigate topics (much like PCA components)

Other NLP Features

- Part of Speech tags
- Dependency Parsing
- Entity Detection
- Word Vectors
- See spaCy!

Using spaCy for NLP

Using spaCy for NLP

In [76]:
```
import spacy

# uncomment the line below the first time you run this cell
#%run -m spacy download en_core_web_sm
try:

    nlp = spacy.load("en_core_web_sm")

except OSError as e:
    print('Need to run the following line in a new cell:')
    print('%run -m spacy download en_core_web_sm')
    print('or the following line from the commandline with eods-f20 activated:')
    print('python -m spacy download en_core_web_sm')

parsed = nlp("N.Y.C. isn't in New Jersey.")
'|'.join([token.text for token in parsed])
```

Out[76]:
```
"N.Y.C.|is|n't|in|New|Jersey|."
```

spaCy: Part of Speech Tagging

spaCy: Part of Speech Tagging

In [77]:

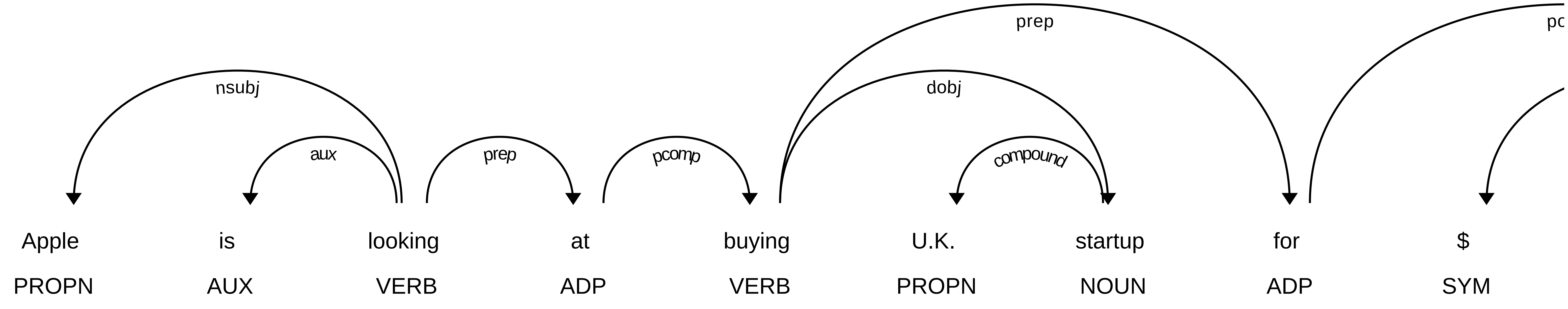
```
doc = nlp("Apple is looking at buying U.K. startup for $1 billion.")

print(f"{'text':7s} {'lemma':7s} {'pos':5s} {'is_stop'}")
print('-'*30)
for token in doc:
    print(f'{token.text:7s} {token.lemma_:7s} {token.pos_:5s} {token.is_stop}')
```

```
text    lemma   pos   is_stop
------------------------------
Apple   Apple   PROPN False
is      be      AUX   True
looking look    VERB  False
at      at      ADP   True
buying  buy     VERB  False
U.K.    U.K.    PROPN False
startup startup NOUN  False
for     for     ADP   True
$       $       SYM   False
1       1       NUM   False
billion billion NUM   False
.       .       PUNCT False
```

spaCy: Part of Speech Tagging

spaCy: Part of Speech Tagging

In [78]:
```
from spacy import displacy
displacy.render(doc, style="dep")
```



spaCy: Entity Detection

spaCy: Entity Detection

In [79]:
```
[(ent.text,ent.label_) for ent in doc.ents]
```

Out[79]:
```
[('Apple', 'ORG'), ('U.K.', 'GPE'), ('$1 billion', 'MONEY')]
```

spaCy: Entity Detection

In [79]:
```
[(ent.text,ent.label_) for ent in doc.ents]
```

Out[79]:
```
[('Apple', 'ORG'), ('U.K.', 'GPE'), ('$1 billion', 'MONEY')]
```

In [80]:
```
displacy.render(doc, style="ent")
```

Apple ORG is looking at buying U.K. GPE startup for $1 billion MONEY .

spaCy: Word Vectors

- word2vec
- shallow neural net
- predict a word given the surrounding context (SkipGram or CBOW)
- words used in similar context should have similar vectors

spaCy: Word Vectors

- word2vec
- shallow neural net
- predict a word given the surrounding context (SkipGram or CBOW)
- words used in similar context should have similar vectors

In [81]:
```
# Need either the _md or _lg models to get vector information
# Note: this takes a while!
#%run -m spacy download en_core_web_md
```

spaCy: Word Vectors

- word2vec
- shallow neural net
- predict a word given the surrounding context (SkipGram or CBOW)
- words used in similar context should have similar vectors

In [81]:
```
# Need either the _md or _lg models to get vector information
# Note: this takes a while!
#%run -m spacy download en_core_web_md
```

In [82]:
```
nlp = spacy.load('en_core_web_md') # _lg has a larger vocabulary

doc = nlp('Baseball is played on a diamond.')
doc[0].text, doc[0].vector.shape, list(doc[0].vector[:3])
```

Out[82]:
```
('Baseball', (300,), [0.55838, 0.42791, -0.11687])
```

spaCy: Multiple Documents

spaCy: Multiple Documents

In [83]:
```
# Use nlp.pipe to transform multiple docs at once
docs = list(nlp.pipe(['Baseball is played on a diamond.',
                      'Hockey is played on ice.',
                      'Diamonds are clear as ice.']))
```

spaCy: Multiple Documents

In [83]:

```
# Use nlp.pipe to transform multiple docs at once
docs = list(nlp.pipe(['Baseball is played on a diamond.',
                      'Hockey is played on ice.',
                      'Diamonds are clear as ice.']))
```

In [84]:

```
# using average of token vectors for each document.
np.array([['{:.2f}'.format(docs[i].similarity(docs[j])) for j in range(3)]
          for i in range(3)])
```

Out[84]:

```
array([['1.00', '0.85', '0.76'],
       ['0.85', '1.00', '0.77'],
       ['0.76', '0.77', '1.00']], dtype='<U4')
```

Learning Sequences

- Hidden Markov Models
- Conditional Random Fields
- Recurrant Neural Networks
- LSTM
- GPT3
- BERT

NLP Review

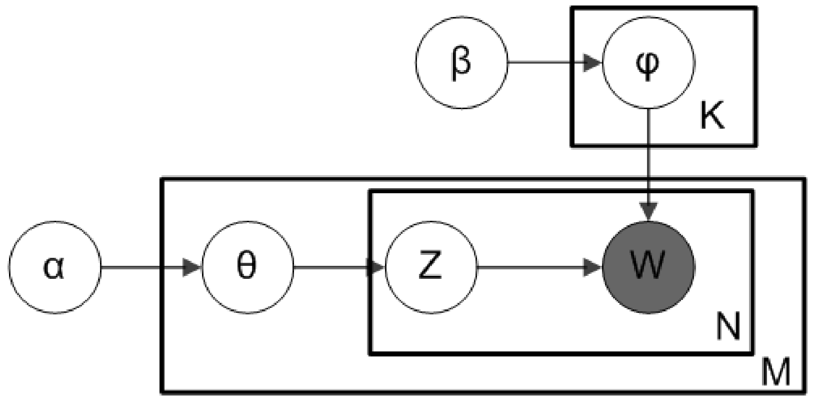
- corpus, tokens, vocabulary, terms, n-grams, stopwords
- tokenization
- term frequency (TF), document frequency (DF)
- TF vs TF-IDF
- sentiment analysis
- topic modeling

- POS
- Dependency Parsing
- Entity Extraction
- Word Vectors

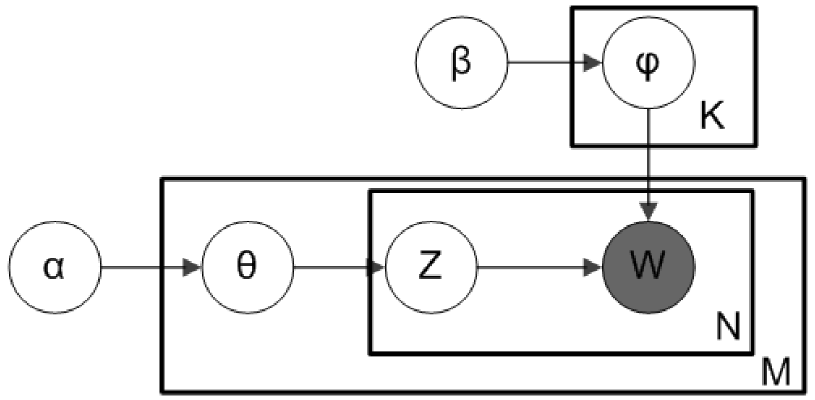
Questions?

Appendix: LDA Plate Diagram

Appendix: LDA Plate Diagram



Appendix: LDA Plate Diagram



K : number of topics

φ : per topic term distributions

β : parameters for word distribution die factory, length = V (size of vocab)

M : number of documents

N : number of words/tokens in each document

θ : per document topic distributions

α : parameters for topic die factory, length = K (number of topics)

z : topic indexes

w : observed tokens