NLP Programmer Databook

Jinghong Chen

2020.9

Contents

Data IO

Data IO concerns reading/saving the data into memory.

pandas

Pandas DataFrame provides a convenient container for reading/manipulating data in `csv` format.

```
import pandas as pd
```

Reading Data

```
df = pd.read_csv('path_to_file')
```

Saving Data

```
df.to_csv('path_to_file', mode='w', encoding='utf_8_sig')
```

Note that `encoding` must be set to `utf_8_sig` to store **Chinese characters**

Convert to native list

```
py_list = df['col'].values.tolist()
```

Making DataFrame

```
some_dict = {k1:v1, k2:v2, ...}
df = pd.DataFrame(some_dict, [index=])
```

The `index` argument can be set to rename the indices. `v` are typically lists of the same length.

json

json is a popular format for storing data in key-value pairs. The data is stored as strings and follows the JavaScript class syntax.

```
import json
```

Reading from json

```
with open('path_to_file', 'r) as f:
    py_dict = json.load(f)
```

The json file is loaded as a **python dictionary**. The information can also be recovered from a json string instead of a file.

```
s is a json string
py_dict = json.loads(s)
```

The `s` is short for `string`.

> Note the difference between `load` and `loads`

Writing to json

There are two common ways to store into a json file.
1. Convert to json string first ,then write to file `json.dumps()` 2. Write to file directly `json.dump()`

```
s = json.dumps(py_dict) # s is a json string
with open('path_to_file', 'w') as f:
    f.write(s)
OR
with open('path_to_file', 'w') as f:
    json.dump(py_dict, f)
```

Data Preprocessing and Filtering

pandas

Pandas is convenient for batch preprocessing and filtering

Iterating by rows

```
for index, row in df.iterrows():
    #do something
```

Batch Processing by Columns

```
def func(x):
    # Preprocessing Routine
df['col'] = df['col'].apply(func)
```

`func` can be defined as lambda as well.

Filter by column value

There are a mainly two ways to do this:

1. `df[cond]` syntax

   filtered_df = df[df['col'] == some_value]

For multiple conditions

```
filtered_df = df[(df['col1'] == some_value) & (df['col2'] == some_value)]
```

And `&`, or `|`

Negation

```
filtered_df = df[~(df['col1'] == some_value)]
```

2. `df.query()` syntax

`df.query()` lets you write the condition in a sql-like fashion. The following commands are identical in effect:

```
filtered_df = df[(df['col1'] == some_value) & (df['col2'] == some_value)]
filtered_df = df.query('col1 == some_value & col2 == some_value')
```

This can be useful in constructing complex filters.

Filter by string

Very often in NLP tasks we need to filter by the existence of certain string pattern, if we describe the pattern by regular expression in raw string `regex`, we can do the following:

```
new_df = df[df['col'].str.contains(regex)]
new_df = df.query(f'col.str.contains({regex})')
```

Handling N/A values

N/A values can occurs when the cell is not filled. Common strategies are:
1. Dropping N/A values `df.dropna(inplace=True)` 2. Filling N/A values `df.fillna(value=<v>, inplace=True)`

re

`re` is the python module for regular expression. For basics refer to official doc. Here we only list a few useful tricks.

Finer Extraction by group

Oftentime, we want to extract texts which obeys certain structure. This can be done by the **grouping syntex** of regular expression.

```
>>> text = 'Number 1 is Eric.'
>>> regex = r'Number (\d+) is (.).'
>>> m = re.search(regex, text)
>>> m.group(1)
1
>>> m.group(2)
Eric
```

Group can be defined by `()` within the regular expression. `m.group(0)` will return the whole matched string. `m` is a match object returned by `re.search()`.

Match by string length

This can be useful in information extraction and preventing greedy matching.

```
re.search('.{5}', text) # match words with exactly 5 words
re.search('.{2,7}', text) # match words with length from 2 to 7 (inclusive)
```

jieba

jieba is a module for Chinese word segementation. Because unlike English, the words in Chinese are not space separated. Segmentation is a necessary preprocessing step before feeding the corpora into algorithms designed for English (such as the `vectorizer` in `sklearn`). Documentation can be found Here. We only gives minimal examples here:

```
import jieba
seglist = jieba.cut(text) # text is string, seglist is list of words

import jieba.posseg as pseg # seg with POS word labels
words = pseg.cut(text)
for word, flag in words:
    # do something
```

Feature Selection

sklearn

We can use functions implemented in `sklearn` to do χ^2 feature selection. A minimal example is given below:

```
from sklearn.feature_selection import SelectKBest, chi2
def chi2_select_K(corpus, y, k):
    vec = CountVectorizer()
    X = vec.fit_transform(corpus)
    chi2_Podel = SelectKBest(chi2, k=k)
    chi2_model.fit_transform(X, y)
    selected_features = [vec.get_feature_names()[i] for i in chi2_model.get_support(indices=
    return selected_features
```

where `corpus` is the corpus (list of lists) and `y` are the corresponding classes of each document.
This simpliest approach uses a $n \times 2$ contigency table to compute χ^2, where each row represents a class and the columns are counts of occurences and non-occurences.

There are two other common methods.
1. Build n contigency tables for each term, calculate the χ^2 as their average
2. Take the top k/n terms for each class.
We omit the specific implementation here.

Feature Transformation

sklearn

sklearn can be helpful in vectorizing the corpora or obtaining dictionary.

`CountVectorizer`

```
from sklearn.feature_extraction.text import CountVectorizer
vec = CountVectorizer([max_df=, min_df=, vocabulary=, stop_words=,])
X = vec.fit_transform(corpus)
vec.vocabulary_ # vocabulary
```

Parameters:

- `max_df`, `min_df` maximum/minimum document frequency to include a word into vocabulary
- `vocabulary` iterable or mappings: specified vocabulary to be used
- `stop_words` words excluded from vocabulary

Each row of `corpus` is a document string.

`TfidfVectorizer`

```
from sklearn.feature_extraction.text import TfidfVectorizer
vec = TfidfVectorizer([max_df=, min_df=, vocabulary=, stop_words=,])
X = vec.fit_transform(corpus)
vec.vocabulary_ # vocabulary
vec.idf_ # idfs for the vocabulary, array
vec.stop_words_ # words ignored
```

The important parameters are identical to `CountVectorizer`

Label Binarizers

LabelBinarizer Binarize labels in a one-vs-all fashion.

```
>>> from sklearn.preprocessing import LabelBinarizer
>>> lb = LabelBinarizer()
>>> lb.fit([1, 2, 6, 4, 2])
>>> lb.classes_
array([1, 2, 4, 6])
>>> lb.transform([1, 6])
array([1, 0, 0, 0],
      [0, 0 ,0, 1])
```

MultiLabelBinarizer Binarize Multi-label problems

```
>>> from sklearn.preprocessing import MultiLabelBinarizer
>>> mlb = MultiLabelBinarizer()
>>> mlb.fit_transform([(1, 2), (3,)])
array([[1, 1, 0],
       [0, 0, 1]])
>>> mlb.classes_
array([1, 2, 3])
```

Building Models

sklearn

sklearn has implemented a range of common machine learning models and is useful for building benchmark.

Pipeline

Pipeline is useful to pack data preprocessing for X (**cannot preprocess y**) and subsequent classifiers. A typical example is:

```
clf = Pipeline([
   ('tfidf', TfidfVectorizer()),
```

```
    ('clf', LinearSVC(class_weight='balanced'))
    ])
clf.fit(corpus, y)
y_hat = clf.predict(new_text)
```

Support Vector Machine

```
from sklearn.svm import LinearSVC
clf = LinearSVC([class_weight={'balanced', None, dict}, dual={True,False}])
```

Parameters:

- `class_weight` sets the weights for data points of each class. If set to `balanced`, the weight are adjusted by their frequency.
- `dual` selects whether to use the dual or primal optimization problem. Prefer `dual=False` when `n_sample > n_feature` and vice versa. Default to `True`

Naive Bayes Classifier

```
from sklearn.naive_bayes import MultinomialNB, BernoulliNB, GaussianNB
```

Multi-layer Perceptrons

```
from sklearn.neural_network import MLPClassifier
clf = MLPClassifier([max_iter=, learning_rate={}, early_stopping={}, activation={}, hidden_l
```

Parameters:

- `max_iter` maximum number of iteration when not converged

- `learning_rate = {‘constant', 'invscaling', 'adaptive'}` Learning rate scheme for weight update.
  - `'constant'` is constant lr
  - `'invscaling'` gradually decreases lr at each time step ‘t’ using inverse scaling exponent ‘power_t’. effective_lr = lr_init/pow(t, power_t)
  - `'adaptive'` keeps lr constant as long as loss is decreasing. And divide by 5 if trainning loss fails to decrease.
- `early_stopping` whether to use early stopping. It will automatically set aside 10% of training data as validation data.
- `activation = {'identity', 'logistic', 'tanh', 'relu'}` the activation function.

One-VS-Rest Classifer

The `OneVsRestClassifier()` wrapper will automatically creates one classifier for each class to tackle **multi-label classification**.

```
from sklearn.multiclass import OneVsRestClassifier
```

```
clf = OneVsRestClassifier(MultinomialNB())
clf.fit(X, Y) # Y could have multi-labels on each row.
```

Metrics

Classification Metrics

1. Confusion Matrix

Confusion Matrix is useful for multi-class classification problems.

```
from sklearn.metrics import confusion_matrix, hamming_loss, jaccard_score
>>> y_true = [2, 0, 2, 2, 0, 1]
>>> y_pred = [0, 0, 2, 2, 0, 2]
>>> conf_mat = confusion_matrix(y_true, y_pred)
array([[2, 0, 0],
       [0, 0, 1],
       [1, 0, 2]])
```

For binary class classification, we can extract true positive etc.

```
>>> tn, fp, fn, tp = confusion_matrix([0, 1, 0, 1], [1, 1, 1, 0]).ravel()
>>> (tn, fp, fn, tp)
(0, 2, 1, 1)
```

2. Hamming Loss

Hamming loss is the fraction of labels that are incorrectly predicted. I.e, the fraction of change that needs to be made to make Y_pred == Y_true.

```
hl = hamming_loss(Y_true, Y_pred)
```

3. Jaccard Score

Defined as the size of intersection over union. I.e, TP/(TP+FN+FP)

```
jac = jaccard_score(Y_true, Y_pred, [average={'micro', 'macro', 'samples', 'weighted', 'bina
```

Parameters:

- `average`
  - `binary` only report results for the class specified by `pos_label`
  - `micro` calculate metrics globally by counting the total TP, FN and FP
  - `weighted` calculate metrics for each label, and find their weighted average
  - `samples` calculate metrics for each instance, and find their average (meaningful for multi-label)

Pytorch

Pytorch is very handy when it comes to customizing neural networks.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

Customized Net Paradigm

Define Model

```
class Net(nn.Module):
'''Implementing Nerual Network in word2vec'''
    def __init__(self, vocab_size, embedding_size):
        super().__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.embeddings.weight = nn.Parameters(<initial tensor>)
        self.embeddings.weight.requires_grad = True
        self.linear = nn.Linear(embedding_dim, vocab_size)

    def forward(self, input, Y):
        embeds = self.embeddings(input)
        out = self.linear(embeds)
        out = F.log_softmax(out, dim=1)
        return out
```

Training

```
model = Net(vocab_size, embedding_size)
optimizer = optim.Adam(model.parameters())
loss_func = nn.NLLLoss()
model.train() # call model.eval() to make sure dropouts work during inference
for e in range(epoch):
    for i in range(total_size//batch_size):
        model.zero_grad()
        out = model(X_batch)
        loss = loss_func(out, Y_batch)
        loss.backward()
        optimizer.step()
```

Data Visualization

sklearn

PCA Decomposition

Principal Components Analysis (PCA) is useful in revealing linear structure. It can be done easily with `sklearn`:

```
from sklearn.decomposition import PCA
```

```
pca = PCA(n_components=2) # reduce to 2-dimensional space
pca_vec = pca.fit_transform(X)
plt.scatter(pca_vec[:,0], pca_vec[:,1])
for w, x, y in zip(vocab, pca_vec[:,0], pca_vec[:,1]):
    plt.annotate(w, (x,y))
plt.show()
```

The above example is taken from visualizing word vectors, hence the annotation.

tSNE

tSNE is useful in revealing maniford structure, i.e., non-linear geometry such as sphere etc. The code is very much the same as PCA.

```
from sklearn.manifold import TSNE
tsne = TSNE()
tsne_vec = tsne.fit_transform(X)
... (AS PCA)
```

seaborn

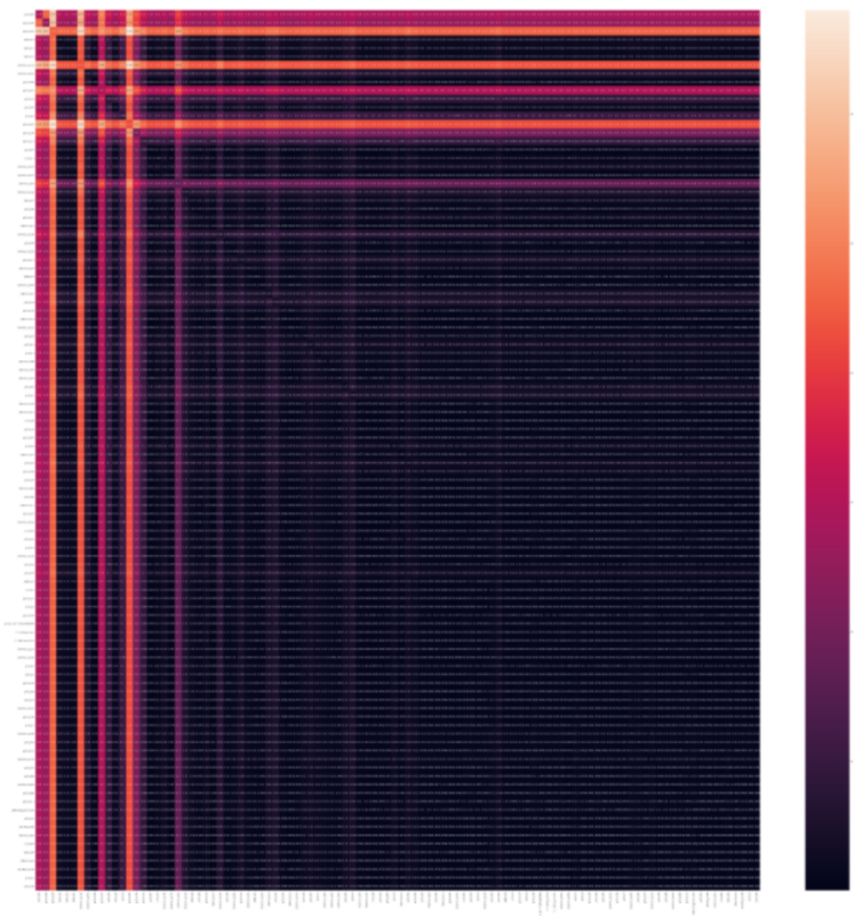
Seaborn is a package for easier drawing. It builds on matplotlib.

```
import seaborn as sns
```

Heat Map

```
matrix_df is a dataframe obtained from the matrix to visualize
hm = sns.heatmap(matrix_df, annot=True)
```

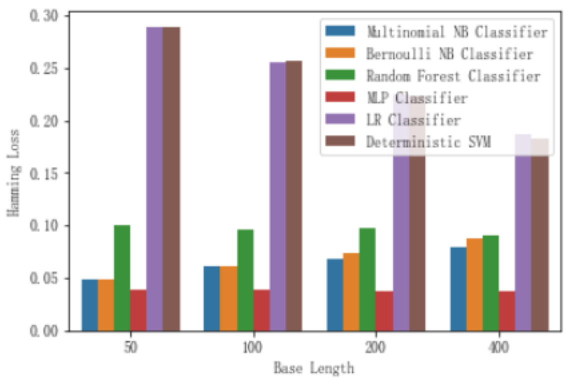
`annot=True` will annotate the value on each cell.



Bar Chart

```
result_dict = {'Base Length:':[...], 'Hamming Loss':[...], 'Name':[...]}
ax = sns.barplot(x='Base Length', y='Hamming Loss', hue='Name', data= result_dict)
```

`x`, `y` and `hue` are the **keys** of the data dictionary.



Chinese Character Display

Matplotlib does not support Chinese character display out of the box. You need to specify the **path to Chinese font file**. For example:

```
from matplotlib.font_manager import FontProperties
zh_font = FontProperties(fname='path_to_ttc')
plt.annotate(w, (x,y), fontproperties=zh_font)
```

On Linux, this font file is usually under `/usr/share/fonts` This can also be done by:

```
from pylab import mpl
mpl.rcParams['font.sans-serif'] = ['FangSong']
mpl.rcParams['axes.unicode_minus'] = False
```

The approaches should work on Windows and Linux.