Python Codebook

greatlearning
Learning for Life

Preface

Data Science is the art and science of solving real world problems and making data driven decisions. It involves an amalgamation of three aspects and a good data scientist has expertise in all three of them. These are:

1) Mathematical/ Statistical understanding
2) Coding/ Technology understanding
3) Domain knowledge

Your lack of expertise should not become an impediment in your journey in Data Science. With consistent effort, you can become fairly proficient in coding skills over a period of time. This Codebook is intended to help you become comfortable with the finer nuances of Python and can be used as a handy reference for anything related to data science codes throughout the program journey and beyond that.

In this document we have followed the following syntax:

- Brief description of the topic
- Followed with a code example.

Please keep in mind there is no one right way to write a code to achieve an intended outcome. There can be multiple ways of doing things in Python. The examples presented in this document use just one of the approaches to perform the analysis. Please explore by yourself different ways to perform the same thing.

Contents

Text Mining

Text Analysis is a major application field for machine learning algorithms. However, the raw data, a sequence of symbols cannot be fed directly to the algorithms themselves as most of them expect numerical features with a fixed size rather than the raw text documents with variable length.
source: scikit-learn
Most of the data in the real world is unstructured text data and the method of mining this unstructured data or pre-processing the text data to get useful insights is called Text Mining Analytics.
A few terminologies used in Text Mining:

1. Bag of Words: Simplification of text. Disregarding grammar.
2. Corpus: A large set of text.
3. Stop Words: Common words which are not useful for deriving meaningful insights. E.g. Articles or Prepositions
4. Stemming: Different variations of a word are changed into the original root word. E.g. Chopped and Chopping is changed to Chop
5. Term Document Matrix (TDM): A matrix which contains the occurrence of the number of terms in each document.
6. Document Term Matrix (DTM): Transpose of TDM.
7. Term Frequency (TF): Normalized count of terms occurring in each document.
8. Inverse Document Frequency (IDF) - To be put very simply, IDF penalizes the term that occurs in almost every document. E.g. “a , an, the”.
9. Lexicon: List of words
10. Bigrams: Collection of words taken two at a time

To analyse the text data, we can start by removing the stop words and then go on to stem words as well. Also, removing punctuation might be a good idea.

from nltk.corpus import stopwords
‘iterative variable’ for ‘iterative variable’ in ‘variable which contains text data’ if not word in stopwords.words()

For stemming purposes, we can use various stemmers that are present in Python. The following is an example of Porter Stemmer

```
from nltk.stem import PorterStemmer

from nltk.tokenize import word_tokenize


ps = PorterStemmer()


sentence = "Programers program with programing languages"

words = word_tokenize(sentence)


for w in words:

    print(w, " : ", ps.stem(w))
```

```
Programers : program

program : program

with : with

programing : program

languages : language
```

You will notice that for stemming we have tokenized the data. A tokenizer is simply a function that breaks a string into a list of words. In the following code snippet, we are removing the punctuation marks from a document of text.

```
import nltk

nltk.download('punkt')

from nltk.tokenize import RegexpTokenizer

tokenizer = RegexpTokenizer(r'\w+')

result = tokenizer.tokenize('hey! how are you ? buddy')

print(result)
```

```
['hey', 'how', 'are', 'you', 'buddy']
```

If we are to count the number of vectors, the following is an in-built function in sklearn. Do refer to the sklearn documentation to understand more about the function

Source: scikit-learn

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer()

corpus = [

'This is the first document.',

'This is the second document.',

'And the third one.',

'Is this the first document?',

]

X = vectorizer.fit_transform(corpus)

print(vectorizer.get_feature_names())
```

```
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
```

Important Libraries

Now, let us try to understand the functionalities of TF-IDF.

TF * IDF gives us a value for a particular word which tells us the significance of that word in the corpus.

```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer (max_features=2500, min_df=7, max_df=0.8, stop_words=stopwords.words('english'))

processed_features = vectorizer.fit_transform(processed_features).toarray()
```

Two things to check

(i) Processed features should be included before this code
(ii) In processed_features, feature should be mentioned as a sample.

Do look for the documentation of the TfidfVectorizer function in sklearn to learn more about the parameters that can be passed for this function.
Source: scikit-learn

After building the TF-IDF, you have successfully managed to convert that unstructured data into structured numeric data and the data can now be used for various Unsupervised or Supervised Learning problems.

With nltk (Natural Language Toolkit) and sklearn, we can also use regular expressions to get meaning out of our text data.

Following are just a few examples of regular expressions:

```
processed_features = [ ]


for sentence in range(0, len(features)):  #here the unstructured has been saved in the variable 'features'
    # Remove all the special characters
    processed_feature = re.sub(r'\W', ' ', str(features[sentence]))


    # remove all single characters
    processed_feature= re.sub(r'\s+[a-zA-Z]\s+', ' ', processed_feature)


    # Remove single characters from the start
    processed_feature = re.sub(r'\^[a-zA-Z]\s+', ' ', processed_feature)


    # Substituting multiple spaces with single space
    processed_feature = re.sub(r'\s+', ' ', processed_feature, flags=re.I)


    # Removing prefixed 'b'
    processed_feature = re.sub(r'^b\s+', '', processed_feature)


    # Converting to Lowercase
    processed_feature = processed_feature.lower()


    processed_features.append(processed_feature)
```

Before executing these code snippets, you have to import the regular expression library by running the following code snippet 'import re'.

To plot a Word Cloud, refer to the following code snippet. The biggest words in the Word Cloud are the words which occur the most number of times. Do remember to remove the stop words and perform stemming of the words before the word cloud as that will help you to get a better idea of the word occurring the most number of times correctly. Sometimes, word clouds are plotted without stemming the words as well.

```
stop_words = set(stopwords.words('english'))  #intialise stopwords from English Language

filtered_sentence = [] #empty list
```

```
for i in processed_features: # iterating in processes features through each sentence
    word_tokens = word_tokenize(i) # converting each sentence to a token
    for w in word_tokens:#in each token, removing stopwords from english language
        if w not in stop_words:
            filtered_sentence.append(w) #appending non-stopwords to filtered_sentence list
comment_words = ' ' #empty string
stop_words = set(STOPWORDS) #stopwords from Wordcloud

for words in filtered_sentence:
    comment_words = comment_words + words + ' ' #converting to string

wordcloud = WordCloud(width = 1000, height = 1000, #wordcloud image creation
                background_color ='white',
                stopwords = stop_words,
                min_font_size = 10).generate(comment_words)

# plot the WordCloud image
plt.figure(figsize = (8, 8), facecolor = None)
plt.imshow(wordcloud)
plt.axis("off")
plt.tight_layout(pad = 0)
plt.show()
```

The above code snippets are just some ways to preprocess the text data. By no means, they are the only means or ways to process unstructured text data.

Time Series Forecasting

In this particular course, we deal with data which has some time stamps associated with it. We are going to see various tests and techniques for predicting the future data based on the past data.

First let us see the syntaxes of Exponential Smoothing and understand how to code that in Python:

1. Simple Exponential Smoothing (SES)

```
from statsmodels.tsa import holtwinters as hw
build = hw.SimpleExpSmoothing('name of the time series').fit() #for building the model

predict = build.forecast(steps='for how long do you want to predict using this model') #for predicting using the model built
```

Note: You can also select the value of 'alpha' manually while invoking the '.fit()' function.

2. Holt's Exponential Smoothing

```
from statsmodels.tsa.holtwinters import Holt
build = Holt('name of the time series').fit() #for building the model

predict = build.forecast(steps='for how long do you want to predict using this model') #for predicting using the model built
```

Note: You can also select the value of 'alpha' and 'beta' manually while invoking the '.fit()' function.

3. Holt-Winters Exponential Smoothing

```
from statsmodels.tsa.holtwinters import ExponentialSmoothing
build = ExponentialSmoothing ('name of the time series', trend='additive' , seasonal='additive').fit()
#for building the model. Here you have to mention the type of trend and seasonal components how you see fit for exponential
#smoothing. We have chosen additive. But do mention these according to the data at hand.

predict = build.forecast(steps='for how long do you want to predict using this model') #for predicting using the model built
```

Note: You can also select the value of 'alpha', 'beta', 'gamma' manually while invoking the '.fit()' function.

Statsmodels link for Exponential Smoothing-
https://www.statsmodels.org/stable/examples/notebooks/generated/exponential_smoothing.html

Now let us check how to code up the ARIMA function in Python with appropriate syntaxes:

4. To plot Auto Correlation Function (ACF) and Partial Auto Correlation Functions (PACF):

    a. ACF Plot

```
from statsmodels.graphics.tsaplots import plot_acf
plot_acf('name of the time series',  ax=plt.gca())
plt.show()
```

    b. PACF Plot

```
from statsmodels.tsa.stattools import plot_pacf
plot_pacf('name of the time series', ax=plt.gca())
plt.show()
```

Note: You can also pass different parameters to in the plot of ACF and PACF to get the desired specific output.

5. To calculate the Moving Average (MA):

```
'name of the time series'.rolling(window='order of the moving average').mean()
```

6. To check for the stationarity of the series:

   a. Augmented Dickey Fuller (ADF) Test

```
from statsmodels.tsa.stattools import adfuller
adfuller('name of the time series')
```

Statsmodels link for Augmented Dickey-Fuller Test:
https://www.statsmodels.org/stable/generated/statsmodels.tsa.stattools.adfuller.html

   b. Kwiatkowski-Phillips-Schmidt-Shin (KPSS) Test

```
from statsmodels.tsa.stattools import kpss
kpss('name of the time series')
```

Statsmodels link for KPSS Test
: https://www.statsmodels.org/stable/generated/statsmodels.tsa.stattools.kpss.html

7. To calculate the Seasonal Auto Regressive Integrated Moving Average (SARIMA) using the auto arima functionality which looks to return us the best model which has the minimum Akaike Information Criteria (AIC) on the training data:

```
import itertools

# Define the p, d and q parameters to take any value between 0 and 2
p = d = q = range(0, 2)

# Generate all different combinations of p, d and q triplets
pdq = list(itertools.product(p, d, q))

# Generate all different combinations of seasonal p, q and q triplets
seasonal_pdq = [(x[0], x[1], x[2], 'value of the seasonality in the SARIMA model') for x in list(itertools.product(p, d, q))]


#Initializing the looping parameters

import numpy as np

best_aic = np.inf
best_pdq = None
```

```
best_seasonal_pdq = None
temp_model = None


#Loop function to calculate the auto arima


for param in pdq:
   for param_seasonal in seasonal_pdq:
      try:
         temp_model = sm.tsa.statespace.SARIMAX('name of the time series',
                                  order = param,
                                  seasonal_order = param_seasonal,
                                  enforce_stationarity=True)
         results = temp_model.fit()
         if results.aic < best_aic:
            best_aic = results.aic
            best_pdq = param
            best_seasonal_pdq = param_seasonal
      except:
         #print("Unexpected error:", sys.exc_info()[0])
         continue
print("Best SARIMA{}{} model - AIC:{}".format(best_pdq, best_seasonal_pdq, best_aic))
```

Now that we have got the best seasonal parameters for the SARIMA, let us build the SARIMA model.

```
Import statesmodels.api as sm
best_model = sm.tsa.statespace.SARIMAX('name of the time series',
                        order=(p, d, q), [(p.d.q) values got from the above loop]
                        seasonal_order=(P,D,Q,m), [(P,D,Q) values got from the above loop and m is the seasonal parameter])
best_results = best_model.fit() #building the model
best_results.forecast()#predicting using the model built


#To check the diagnostics of the model built
best_results.plot_diagnostics(lags='desired lags to be specified')
plt.show()
```

Note: You can also use the different parameters in the SARIMAX function to get the desired output of ARIMA, SARIMA and SARIMAX. Do refer to the SARIMAX documentation in the statespace submodule of statsmodels library.

https://www.statsmodels.org/dev/generated/statsmodels.tsa.statespace.sarimax.SARIMAX.html