Exploratory Data Analysis

DS 8015

Outline

1. Pandas
2. Distribution Analysis
3. Correlation identification

Pandas

Object Creation - 1

```python
import numpy as np
import pandas as pd
print(pd.__version__) # print pandas version

#Creating a Series by passing a list of values,
# letting pandas create a default integer index:
s = pd.Series([1,3,5,np.nan,6,8])

#Creating a DataFrame by passing a NumPy array,
# with a datetime index and labeled columns:
dates = pd.date_range('20130101', periods=6)
df = pd.DataFrame(np.random.randn(6, 4), index=dates,
   columns=list('ABCD'))
```

Object Creation - 2

```
#Creating a DataFrame by passing a dict of objects
#  that can be converted to series-like.
df2 = pd.DataFrame({
'A': np.random.randint(0,10, size=5),
'B': pd.Series(2, index=list(range(5)), dtype='float32'),
'C': np.array([3]*5, dtype='int32'),
'D': pd.Categorical(["test", "train", "test", "train", "train"]),
'E': 'foo',
'F': pd.date_range(end='20131231', periods=5)})

# set index of the dataframe
df2.index = pd.date_range(start='20130101', periods=5)
print(df2.dtypes)
# A int64
# B float32
# C int32
# D category
# E object
# F datetime64[ns]
```

Viewing Data

```
# View the top/bottom rows of the frame:
df.head()
df.tail()
# Display the index, columns:
df.index
df.columns

# DataFrame.to_numpy() gives a NumPy representation of the data.
df.to_numpy()

# Show a quick statistic summary of your (numerical) data:
df.describe() # df.describe(include="all")

# transposing the data
df.T

# Sorting
df.sort_index(axis=1, ascending=False) # by axis
df.sort_values(by='B') # by values
```

SELECTION

```
#We will see optimized pandas data access methods
# .at, .iat, .loc and .iloc.

# Selecting a single column, which yields a Series:
df['A']
# df.A

# Selecting via [], which slices the rows.
df[0:3]
df['20130102':'20130104']
```

Selection by label

```
# For getting a cross section using a label:
df.loc[dates[0]]

# Selecting on a multi-axis by label:
df.loc[:, ['A', 'B']]

# Showing label slicing, both endpoints are included:
df.loc['20130102':'20130104', ['A', 'B']]

# Reduction in the dimensions of the returned object:
df.loc['20130102', ['A', 'B']]

# For getting a scalar value:
df.loc[dates[0], 'A']

#For getting fast access to a scalar
# (equivalent to the prior method):
df.at[dates[0], 'A']
```

SELECTION BY POSITION

```
# Select via the position of the passed integers:
df.iloc[3]
df.iloc[3][1]
df.iloc[3]['B']

# By integer slices, acting similar to numpy/python:
df.iloc[3:5, 0:2]

# By lists of int pos. loc., similar to the numpy/python style:
df.iloc[[1, 2, 4], [0, 2]]

# For slicing rows/columns explicitly:
df.iloc[1:3, :]
df.iloc[:, 1:3]

# For getting a value explicitly:
df.iloc[1, 1]

# For getting fast access to a scalar (equiv. to prior method):
df.iat[1, 1]
```

Boolean Indexing

```
# Using a single column's values to select data.
df[df.A > 0]

#Selecting values from a DataFrame
# where a boolean condition is met.
df[df > 0]

# Using the isin() method for filtering:
df2 = df.copy()
df2['E'] = ['one', 'one', 'two', 'three', 'four', 'three']
df2[df2['E'].isin(['two', 'four'])]
```

SETTING

```
# Setting a new column automatically aligns data by indexes.
s1 = pd.Series([1, 2, 3, 4, 5, 6],
   index=pd.date_range('20130102', periods=6))
df['F'] = s1

# Setting values by label:
df.at[dates[0], 'A']=0

# Setting values by position:
df.iat[0,1]=0

# Setting by assigning with a NumPy array:
df.loc[:, 'D'] = np.array([5] * len(df))

# A where operation with setting.
df2 = df.copy()
df2[df2 > 0] = -df2
```

MISSING DATA

- pandas primarily uses the value np.nan to represent missing data.
- It is by default not included in computations.
- Reindexing allows you to change/add/delete the index on a specified axis. This returns a copy of the data.

```
df1 = df.reindex(index=dates[0:4],columns=list(df.columns)+['E'])
df1.loc[dates[0]:dates[1], 'E'] = 1

# To drop any rows that have missing data.
df1.dropna(how='any')

# Filling missing data.
df1.fillna(value=10)

# To get the boolean mask where values are nan.
pd.isna(df1)
```

OPERATIONS - 1

- Operations in general exclude missing data.

```
# Performing a descriptive statistic:
df.mean()

# Same operation on the other axis:
df.mean(1)

#Subtraction of dataframe and other,
# element-wise (binary operator sub).
s = pd.Series([1, 3, 5, np.nan, 6, 8], index=dates).shift(2)
df.sub(s, axis='index')

# applying functions to the data
df.apply(np.cumsum)
df.apply(np.cumsum, axis=1)
df.apply(lambda x: x.max() - x.min())
```

Operations - 2

```
# histogramming
s = pd.Series(np.random.randint(0, 7, size=10))
s.value_counts()
s.hist() # bins=10

# string methods
s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan,
   'CABA', 'dog', 'cat'])
s.str.lower()

# Concatenating pandas objects together with concat():
df = pd.DataFrame(np.random.randn(10, 4))
# break into pieces
pieces = [df[:3], df[3:7], df[7:]]
pd.concat(pieces)
```

OPERATIONS - 3

```
# join: for SQL style merges
left = pd.DataFrame({'key': ['foo', 'foo'], 'lval': [1, 2]})
right = pd.DataFrame({'key': ['foo', 'foo'], 'rval': [4, 5]})
pd.merge(left, right, on='key')

left = pd.DataFrame({'key': ['foo', 'bar'], 'lval': [1, 2]})
right = pd.DataFrame({'key': ['foo', 'bar'], 'rval': [4, 5]})
pd.merge(left, right, on='key')

# Append rows to a dataframe
df = pd.DataFrame(np.random.randn(8, 4),
   columns=['A', 'B', 'C', 'D'])
s = df.iloc[3]
df.append(s, ignore_index=True)
```

GROUPING

- By "group by" we are referring to a process involving one or more of the following steps:
  - (1) Splitting the data into groups based on some criteria
  - (2) Applying a function to each group independently
  - (3) Combining the results into a data structure

```
df = pd.DataFrame({
'A': ['foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'foo'],
'B': ['one', 'one', 'two', 'three', 'two', 'two', 'one', 'three'],
'C': np.random.randn(8),
'D': np.random.randn(8)
})

#Grouping and then applying the sum()
# function to the resulting groups.
df.groupby('A').sum()

#Grouping by multiple columns forms a hierarchical index,
# and again we can apply the sum function.
df.groupby(['A', 'B']).sum()
```

Reshaping

```
# pivot tables
df = pd.DataFrame({
'A': ['one', 'one', 'two', 'three'] * 3,
'B': ['A', 'B', 'C'] * 4,
'C': ['foo', 'foo', 'foo', 'bar', 'bar', 'bar'] * 2,
'D': np.random.randn(12),
'E': np.random.randn(12)
})

pd.pivot_table(df, values='D', index=['A', 'B'], columns=['C'])
```

CATEGORICALS

```
# pandas can include categorical data in a DataFrame
df = pd.DataFrame({"id": [1, 2, 3, 4, 5, 6],
                   "raw_grade": ['a', 'b', 'b', 'a', 'a', 'e']})

# Convert the raw grades to a categorical data type.
df["grade"] = df["raw_grade"].astype("category")

#Rename the categories to more meaningful names
# (assigning to Series.cat.categories is inplace!).
df["grade"].cat.categories = ["very good", "good", "very bad"]

#Reorder categories and simultaneously add missing categories
# (methods under Series .cat return a new Series by default).
df["grade"] = df["grade"].cat.set_categories(
["very bad", "bad", "medium", "good", "very good"])

# Sorting is per order in the categories, not lexical order.
df.sort_values(by="grade")
# Grouping by a categorical column also shows empty categories.
df.groupby("grade").size()
```

Plotting

```
ts = pd.Series(np.random.randn(1000),
   index=pd.date_range('1/1/2000', periods=1000))
ts = ts.cumsum()

#On a DataFrame, the plot() method is a convenience to plot
# all of the columns with labels:

df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index,
   columns=['A', 'B', 'C', 'D'])
df = df.cumsum()
plt.figure()
plt.legend(loc='best')

df.plot()
```

Getting Data In/Out

```
# Writing/reading to/from a csv file.
df.to_csv('foo.csv')
pd.read_csv('foo.csv')

# Writing/reading to/from an excel file.
import openpyxl
df.to_excel('foo.xlsx', sheet_name='Sheet1')

pd.read_excel('foo.xlsx', 'Sheet1', index_col=None,
   na_values=['NA'])
```

Distribution Analysis

VISUAL CHECK - 1

```
#Reading the dataset in a dataframe using Pandas
df = pd.read_csv("LoanPrediction/train.csv")

# categorical cols
df.select_dtypes(exclude=['int', 'float']).columns
# Index(['Loan_ID', 'Gender', 'Married', 'Dependents',
#'Education', 'Self_Employed', 'ApplicantIncome', 'Property_Area',
# 'Loan_Status'], dtype='object')

# numerical cols
df.select_dtypes(include=['int', 'float']).columns
#Index(['CoapplicantIncome', 'LoanAmount', 'Loan_Amount_Term',
#'Credit_History'], dtype='object')

#For the non-numerical values, look at frequency distribution
# using following command:
df['Property_Area'].value_counts()
# Semiurban 233, Urban 202, Rural 179
```

VISUAL CHECK - 2

```
#can be used for getting various info about data
# (e.g. existence of outliers)

# plotting the histogram of ApplicantIncome
df['ApplicantIncome'].hist(bins=50)

# look at box plots to understand the distributions
df.boxplot(column='ApplicantIncome')
# segregate ApplicantIncome by Education
df.boxplot(column='ApplicantIncome', by = 'Education')
```

Categorical variable analysis

```
# Show the counts of observations in each categorical bin.
import seaborn as sns
sns.countplot(x="Credit_History", hue="Loan_Status",
   data=df, palette="Greys")

# Compute a simple cross-tabulation of two (or more) factors.
pd.crosstab(index=df['Credit_History'],
   columns=df['Loan_Status'], margins=False)

pd.crosstab(index=[df['Credit_History'],df['Education']],
   columns=df['Loan_Status'], margins=False)

# Stacked chart
temp3 = pd.crosstab(df['Credit_History'], df['Loan_Status'])
temp3.plot(kind='bar', stacked=True, color=['red','blue'],
   grid=False)
```

TREATING EXTREME VALUES

```
# One approach is to remove rows with outliers
df_red = df
percentile_val = df_red.LoanAmount.quantile(0.975)
# reduce df by removing rows based on percentile_val
df_red = df_red[df_red['LoanAmount'] < percentile_val]

# instead of treating extreme values as outliers, use log
df['LoanAmount_log'] = np.log(df['LoanAmount'])

# scale back the extreme values
df["LoanAmount_new"] = df["LoanAmount"]
df["LoanAmount_new"][df["LoanAmount_new"]>percentile_val]
   = percentile_val

# what happens if you do the following?
df[df["LoanAmount_new"]>percentile_val] = percentile_val
```

Correlation identification

PREPROCESSING AND PLOTTING - 1

```
#many correlation functions such as pearson
# and spearman tests require numerical features
# so, first convert categorical feats to numeric ones

# Married is ['No', 'Yes']
df['Married'] =df['Married'].astype('category').cat.codes
# Now, married is [0, 1]

# check the scatter plots
df.plot.scatter(x = 'ApplicantIncome',
   y = 'CoapplicantIncome', c='DarkBlue')
df.plot.scatter(x = 'ApplicantIncome',
   y = 'CoapplicantIncome', c='DarkBlue', , s=df['Dependents'])
```

PREPROCESSING AND PLOTTING - 2

```
# combined scatter plot matrix
from pandas.plotting import scatter_matrix
attributes = ['ApplicantIncome','CoapplicantIncome','LoanAmount']
scatter_matrix(df[attributes], figsize=(12, 8))

# seaborn pairplot
df_new = df[['ApplicantIncome', 'CoapplicantIncome',
 'LoanAmount', 'Loan_Status']]
sns.pairplot(df_new, hue='Loan_Status')

# seaborn scatterplot
sns.scatterplot(x="TotalIncome_log", y="LoanAmount_log",
   hue="Loan_Status", data=df, palette="colorblind")
```

Correlation tests

```
#pearson correlation - used for numeric feats
corr_pearson = df.corr('pearson')
corr_pearson.style.background_gradient(cmap='coolwarm')

#Spearman Rank correlation
# requires ranking the data first
corr_spearman = df.corr('spearman')

#Kendall correlation
# requires ranking the data first
corr_kendall = df.corr('kendall')

#To only obtain the correlation between a
# feature and a subset of the features
df[['ApplicantIncome', 'Education', 'LoanAmount']]
   .corr()['LoanAmount'][:]
```