Pandas and Statsmodels

The pandas package provides very efficient data structures and tools for analyzing data. There are two basic data types created by pandas: `Series` and `DataFrame`. The `Series` class is built to store a column of data. That is, one characteristic and multiple observations. The `DataFrame` class is built to store several columns of related data. We will learn how to work with both classes. Pandas is built-on top of NumPy, and there are several other packages that rely on Pandas for data management. Later, we will work with the Statsmodels package, which provides functions for statistical estimation.

1 Install

We will install Pandas, but also two other packages, Statsmodels and Requests. On the terminal, execute:

```
pip install requests
pip install pandas
pip install statsmodels==0.10.0rc2 --pre
```

The third line in the code above is to install the newest version of Statsmodels that has not yet been fully released. We need to do so, because the currently released version has some incompatibility issues with the most recent SciPy. If you are having issues with the installation process check the pandas install page or the statsmodels install page.

Pandas should now be available under the `pandas` namespace. Just like with NumPy, Pandas is often imported as `pd`. Test if `pandas` is working properly on your computer:

- Start the Python REPL and do `import pandas`;
- If it is imported, then you are good to go.

If you see an error when importing pandas, see this discussion page and this page on how to fix the issue. Alternatively, uninstall pandas with `pip uninstall pandas` and install the previous version with `pip install pandas==0.24`.

2 Series

We start by creating a Pandas Series. The Series object is like a single column of a spreadsheet. Each row of the column is one observation of a single characteristic. Below, we will create a Series to hold the last return of a Stock, and each row will represent the company name:

```
import numpy as np
import pandas as pd

s = pd.Series(data=np.random.random(5), name='returns')
print(s)
```

You can think of each row as a return observation for a stock. Or each row as representing a different company, and the characteristic is the return for a month. The Series object can only hold data of a single type. The values of the column are stored in a numpy array:

```
print(s.values, type(s.values))
```

However, `s` itself is another class with more features. And only one of its attributes is the numpy array.

```
type(s)
```

Mathematical operations work as they would with a Numpy array:

```
s + 1
s * 100
np.exp(s)
np.log(s)
```

The index of the series (the column of numbers) serves as a way to find the data we want. But, we can change the way the data is indexed. For example, we can change from numbers to text:

```
print(s.index)
s.index = ['SPY', 'AAPL', 'TSLA', 'AMZN', 'COST']
print(s)
print(s.index)
```

Notice that now the type of the index is `object`, instead of `float_`. We can access the rows using the same notation as dictionaries:

```
s['SPY']
s['AAPL']
```

But they are more flexible. For example, we can acess multiple rows:

```
s[['SPY', 'AAPL']]
s[['SPY', 'AAPL', 'COST']]
```

We can also use the slicing notation:

```
s[:]
s[0:2]
s[-2:]
s[[0, 1, 3]]
```

We can use the in operator to test against the series index:

```
print('SPY' in s)
print('GOOG' in s)
```

3 DataFrame

A `DataFrame` is a collection of many `Series`. It is like a spreadsheet, where each column represents one variable and each row an observation.

```
df = pd.DataFrame(data=np.random.random((5, 3)))
# DataFrame does not take a name argument.
type(df)
print(df)
```

The rows are indexed by numbers:

```
print(df.index)
df.index = ['SPY', 'AAPL', 'TSLA', 'AMZN', 'COST']
df
```

The columns are also indexed by numbers, but we can also change the indexing to represent the characteristics:

```
print(df.columns)
df.columns = ['return', 'last dividend', 'last price']
df
```

For a DataFrame there are several ways of indexing. However, we need to be careful, since the usual slicing is not valid anymore. The use of brackets is used to access **columns**:

```
df['return']
df['last price']
df[['last dividend', 'return']]
```

Notice that the DataFrame columns are reordered depending on the order of the columns.

Now, to slice rows we use the method `iloc`. You can also use `[]` to slice rows, but that is not the preferred way and might be confusing, since `[]` is mainly used for getting columns.

```
df.iloc[1:3, :2]
df.iloc[:, -2:]
df.iloc[[0, 2], [0, 2]]
```

The method `iloc` is for selectin rows and columns by **integer** indexing. We can use the `loc` method for selecting rows and columns by **label**. Labels are the values given to the indices and columns of the `DataFrame`:

```
df.loc[['AAPL', 'TSLA'], ['return', 'last price']]
df.loc['AAPL', 'return']
df.loc['AAPL', :]
# Get the labels from the index
df.index[:2]
df.loc[df.index[:2], ['return', 'last dividend']]
# Notice that when we select a single column, we get back a
   Series:
type(df.loc[['AAPL', 'TSLA'], 'return'])
# The same happens for a single row:
type(df.loc['AAPL', :])
```

4 Reading Data

Pandas also implements several methods for reading data into a `DataFrame`. Let's use `pd.read_csv` to download .csv data and load it into a DataFrame. `pd.read_csv` can read a local file, but can also take a url to read an online file:

```
url = 'https://raw.githubusercontent.com/python-for-economists/
   lecture-notes/master/supporting/data/business.csv'
data = pd.read_csv(url)
print(data)
```

There are several columns that in the data that we won't use. We can select the only ones we want by indexing:

```
data.columns
data.columns[5:]
columns_to_keep = ['Economy Name', *list(data.columns[5:])]
columns_to_keep
data = data[columns_to_keep]
print(data)
```

Let's change the country name of Korea to South Korea:

```
data.iloc[1, 0] = 'South Korea'
print(data)
```

Let's modify the index, so that instead of using numbers, we use the country names:

```
data.set_index('Economy Name')
# a new object is created when we call set_index and a new
   DataFrame is returned
# the original DataFrame was not modified
data
# But we do want to modify it:
data = data.set_index('Economy Name')
data
data.index
```

Let's change the name of some columns:

```
data.columns = ['total_tax', 'profit_tax',
                'tax_score', 'electricity_score', 'law_score']
data
```

Let's convert the scores to numbers between 0 and 1:

```
data['tax_score'] = data['tax_score']/100
data['electricity_score'] = data.electricity_score/100
data.law_score = data.law_score/100
```

Let's create a column with an equally weighted score based on the scores for taxation, electricity and law enforcement. There are a few ways of doing so:

```
first = (data.tax_score + data.electricity_score + data.
   law_score)/3
second = np.sum(data.iloc[:, 2:], axis=1)/3
third = data.iloc[:, 2:].mean(axis=1)
all(first == second)
```

```
all(second == third)
data['score'] = data[['tax_score', 'electricity_score', '
   law_score']].mean(axis=1)
data
```

5 Plotting

Pandas is also built on top of Matplotlib. The `Series` and `DataFrame` classes provide methods to automatically plot what is stored in these objects:

Create a bar plot with the scores for each country:

```
ax = data['score'].plot(kind='bar', title='Business Score')
ax.set(ylabel='Score (0 to 1)')
```

You can select the data you want to plot, and even do plot each column on a different axis:

```
ax = data[['score', 'total_tax']].plot(kind='bar', subplots=
   True, title=['Business Score','Total Taxes'])
```

Similar functionality is available for line and scatter plots.

Let's sort the data by the score and re-plot:

```
data = data.sort_values(by='total_tax', ascending=True)
ax = data['total_tax'].plot(kind='bar')
```

6 Time Series

Let's use the module `requests` to obtain financial data from the Federal Reserve Bank of St. Louis.

```
import requests

# Download US Unemployment Rate
url = 'http://research.stlouisfed.org/fred2/series/UNRATE/
   downloaddata/UNRATE.csv'
response = requests.get(url)
if response.status_code == 200:
    print("Request succeeded.")
```

The content of the csv file is stored in the response:

```
print(response.content)
# The method decode can help:
print(response.content.decode())
```

We can store the contents into a local `csv` file:

```
with open('UNRATE.csv', 'w') as f:
    f.writelines(response.content.decode())
```

Look at the file with an editor. The first row contains the column headers. The first column contains dates for the observations, and we will use it as the index of the DataFrame. The second column contains the unemployment rate.

We can use pd.read$_{csv}$ to load the data in a `DataFrame`:

```
data = pd.read_csv('UNRATE.csv', index_col=0, parse_dates=True)
# If the parse_dates input is True, it tries to parse the
   values in
# the index as dates.
type(data)
data.dtypes
```

We can look a the first rows of the data and at the last rows of the data with the methods `head` and `tail`:

```
data.head()
data.tail()
```

Get some summary statistics:

```
print(data.describe())
```

Change the number of decimal places PANDAS use:

```
pd.set_option('precision', 1)
print(data.describe())
```

Change the column name:

```
data.columns = ['Unemployment Rate']
```

We can plot the time series:

```
ax = data.plot(kind='line')
```

Or only a few years:

```
ax = data['2009':].plot(kind='line')
```

7 Using an API

Some websites offer an API (application programming interface) that we can use to request data. For example, the World Bank makes available several different indicators via their API. The easiest way to work with their API is to go to the indices page, click on the index you are interested in, and then right-click on the `CSV` button and copy the link (an url). Alternatively, you could click on the `EXCEL` button to get the url for downloading the spreadsheet. Having the url, you can request the data in Python by submitting a get request with the `requests` module.

Let's download the GDP per capita series. If you use the download format for `csv` files, you will get back a `.zip` instead. We can deal `.zip` files directly in Python via the built-in `zipfile` module. For more details read this stack post. We will download the Excel file instead:

```
url = 'http://api.worldbank.org/v2/en/indicator/NY.GDP.PCAP.CD?
   downloadformat=excel'
response = requests.get(url)
if response.status_code == 200:
    print("Request succeeded.")
    with open('gdp_capita.xls', 'wb') as f:
        # open with the option write binary, since xls is a
           binary file
        f.write(response.content)
        print("File saved.")
```

We can load Excel files in pandas as easy as csv files. You might need to install the `xlrd` package (`pip install xlrd`).

```
data = pd.read_excel('gdp_capita.xls', sheet_name='Data',
                     skiprows=3, index_col=1)
type(data)
# Recover the GDP per capita for Brazil
data.columns
data.index
gdp_brazil = data.loc['BRA', data.columns[3:]]
type(gdp_brazil)
gdp_brazil
```

Let's plot it:

```
ax = gdp_brazil.plot(kind='line', color='black', grid=True,
                     title="Brazil's GDP per Capita (in US$)")
```

8 Panel Data

We will use Pandas to work with panel data. The OECD makes available data on real minimum wages of several countries. I have downloaded that data and uploaded to Github, we will download it and store it in a pandas DataFrame:

```
url = 'https://raw.githubusercontent.com/python-for-economists/
   lecture-notes/master/supporting/data/oecd_real_wage.csv'
data = pd.read_csv(url, index_col='Time', parse_dates=True)
print(data.head())
```

Because we have panel data, we will see several repeats of the same country, at different time periods. Let's first clean the columns and keep only what really matters.

```
data.columns
```

We do not need both the country name and the code. The column `Series` describes the two types of wages available, but the name is too long. Let's use the column `SERIES` instead. We can keep the `Pay Period` column since it descibes whether wages are annual or hourly. We also need the `Value` column, since that contains the wages. All the other columns can be thrown away.

```
data = data[['Country', 'SERIES', 'Pay period', 'Value']]
```

The proper way to rename columns is with the method `rename`:

```
data.rename(columns={'SERIES': 'Series', 'Value': 'Real Wage'},
            inplace=True)
print(data)
```

We would like to index our data by the time period. We would also like to divide the data by the country, by the type of the series (USD PPPs or USD exchange rates), and by the type of pay period (annual or hourly). Dividing the data into sub-categories allows us to express multi-dimensional data in a 2-D table! To do so, we need to use a `pivot_table`, which basically breaks down columns (or indices) by several categories:

```
data.columns
data.index
```

```
data = data.pivot_table(values='Real Wage',
                        index='Time',
                        columns=['Country', 'Series', 'Pay
                           period'])
print(data)
```

Now we have a table where the index is given by the years, and the columns are split by the country, then by the series, and then by the pay period.

```
type(data.columns)
print(data.columns)
```

A MultiIndex object is now used to store the columns. One of its attributes is `levels`:

```
print(data.columns.levels)
```

The attribute `levels` is a list of lists, where the first element (which is a list) contains the values of the countries, the second list contains the values of the `Series` column, and the third list contains the values of the `Pay period` column. The MultiIndex is a **hierarchical structure**, and the order it follows was the order that was given in the `pivot_table` function. In this case, the labels first in the hierarchy are the `Country` names, then comes the `Series` names, and last the `Pay period`.

We can select columns of the data by using the slicing operator:

```
data['Brazil']
```

The code above goes into the column of the country Brazil, and displays all the values for this country. We go down 1-level in the hierarchy, and now there are only two divisions for columns, the first is for Series and the second for Pay period.

We can get multiple countries as well:

```
data[['Brazil', 'Chile']]
```

But now we still have the Country hierarchy.

We can go deeper in the hierarchy:

```
data.columns.levels
data['Brazil']['PPP']
data['Brazil']['PPP']['Hourly']
```

The slices above can be used to access data in a single columns (or in a single column for each hierarchy level).

To get multiple columns, we use the `loc` method:

```
data.loc[:, ['Brazil', 'Chile']]
type(data.loc[:, ['Brazil', 'Chile']])
```

In pandas, the use of `[]` when slicing defines the columns that we want to get, but only based on the first level of the hierarchy. If we want to go lower in the hierarchy, we need to parentheses (a tuple). A tuple specifies a complete **multi-level** key, where each element of the tuple specifies the keys for one level:

```
data.loc[:, ('Brazil', 'EXR')]
data.loc[:, ('Brazil', 'EXR', 'Annual')]
```

Each element of the tuple represents the key in one of the levels of the hierarchy.

If we also use brackets, then we are defining multiple columns to get at each hierarchy level:

```
data.loc[:, (['Brazil', 'Chile'], ['EXR'])]
```

The : means we want all rows. Then we have a big tuple (....). A tuple has a special meaning when being used with a `MultiIndex`. Each element of the tuple is related to each level of the `MultiIndex`. The first element of the tuple is a list: `['Brazil', 'Chile']`. A list has a special meaning when being used with a `MultiIndex`. It species a list of keys in the same hierarchical level to retrieve. In this case, we are telling `loc` to retrieve the columns named `'Brazil'` and `'Chile'`. The second element of the tuple is a list with one element: `['EXR']`. It tells `loc` that, given the choices for the first hierarchical level, it should get the column named `'EXR'` on the second level.

We can go lower on the hierarchy and also define the keys for the third level:

```
print(data.loc[:, (['Brazil', 'Chile'], ['EXR', 'PPP'], ['
    Annual'])])
```

Another useful method is `stack`. It takes a column name of the `MultiIndex` and shifts it to the row indices. In doing so, it transforms the row index in a `MultiIndex` too!

```
data.head()
data.stack(level='Country').head()
```

Now, the table has a `MultiIndex` for the indices, where the 1st level of the hierarchy is the time period and the second level is the country name.

Calling `stack` without any arguments shifts the lowest hierarchy of the columns. In our case, it would shift the `Pay period`:

```
data.stack().head()
type(data.stack().head().index)
# hierarchy
data.stack().index.names
```

We can also use numbers instead of labels with stack:

```
# Shift Country:
data.stack(level=(0))
# Shift Series
data.stack(level=(1))
# Shift Pay Period
data.stack(level=(2))
# We can shift multiple levels:
data.stack(level=(1, 2))
```

To get a cross-section of the data, we can fix the time period:

```
data.loc['2018'].stack(level=(1, 2))
# now the column is back to being a regular Index
data.loc['2018'].stack(level=(1, 2)).columns
```

If you want to use the countries names as the main index, we can transpose everything:

```
data['2018'].stack(level=(1, 2)).transpose()
```

Let's go further and select only the annual wage and those measured using the purchasing-power parity: We will do this in 2 ways. First, by direct selection of the columns:

```
csec = data.loc[:, (slice(None), ['PPP'], ['Annual'])]
csec.head()
```

The `slice(None)` means that we want to select all of the values of the 1st hierarchical level.

We can now drop the `Series` and `Pay period` levels since they are fixed:

```
csec = csec.droplevel(level=(1, 2), axis=1)
csec
```

Second, using the `xs` method:

```
help(data.xs)
```

We need to pass it the key representing the values. We also need to specify `axis=1`, since we are getting the data from columns. And specify what is the hierarchy because we are dealing with `MultiIndex`:

```
csec = data.xs(key=('PPP', 'Annual'), axis=1, level=('Series',
    'Pay period')).copy()
csec.head()
type(csec.columns)
```

We use the `copy` method because we are going to modify this data. Without the `.copy`, we would be getting back a "view" of the data. Pandas tries to be very efficient, so it does not copy data most of the time. This means that slicing returns a "zoomed in" version of the table (referred to as a "view"). If we try to modify a zoomed in version of the table, we will get a warning, because it could affect the rest of the data. We need to copy the data from the zoomed in version of the table so that we get a new table (different space in memory). And this new table we can modify. This was achieved by using the method copy.

We can check for missing values with the `isnull` method: Returns `True` if `np.NaN` is found, otherwise it returns `False`:

```
csec.isnull()
# Count by column (sum by row):
csec.isnull().sum(axis=0)
```

There are 14 missing values for Germany and 1 for Japan.

Pandas provides a helper function for filling missing values: fillna. We will use the method `'ffill'` to use the last valid value to fill the next missing value:

```
csec['Japan'].fillna(method='ffill')
csec['Japan']
csec['Japan'] = csec['Japan'].fillna(method='ffill')
csec['Japan']
```

For Germany we have too many missing values:

```
csec['Germany']
```

Too many missing values, so we can try going back to the data source and see what happened (maybe an error?) Try to interpolate the data, or simply drop the country. Let's drop the column:

```
csec = csec.drop(labels=['Germany'], axis=1)
'Germany' in csec.columns
```

9 Merge, Join and Concatenate

We will use another data set that contains data on continent names and countries to add a new level to our panel data:

```
url = 'https://raw.githubusercontent.com/python-for-economists/
   lecture-notes/master/supporting/data/continent_countries.csv'
continents = pd.read_csv(url)
continents
```

Pandas provides several methods of combining different DataFrame and Series objects. We will use the `merge` method to merge the panel data to the `continents` DataFrame we just loaded. To learn more about `merge`, and about the `join` and `concatenate` methods read the Method, join and concatenate reference page.

The `continents` and `csec` have an index in common: the country names.

```
csec
continents
```

We can use the country names as a key to merge the two tables, this is known as a one-to-one join.

Let's transform the two DataFrame objects so that the column Country is their only row index:

```
continents.head()
continents = continents.set_index(keys='Country')
continents.head()

csec = csec.transpose()
csec.head()
```

We now have two tables, where both of them have as index a column with Country names. Notice that the indices must be unique for this to work. We want to keep all the columns of the `csec` table (each column represents a year) and add the column with the continent names from `continents`.

The `merge` method takes the inputs:

- left: the first table;
- right: the second table;
- $\text{left}_{\text{index}}$: specifies if we want to use the index from the `left` table as the key to merge the two tables;
- how: specify what keys to use when merging the tables. If we are using multiple keys, we can specify to take the union of the keys, or the intersection. In our case, we will use only the country names from the `csec` tables as the key.
- $\text{right}_{\text{index}}$: specifies if we want to use the index from the `right` table as the key to merge the two tables. In our case, we want this to be True.

The keys that we specify on the two tables are the values that will be used to match the values from the tables. By specifying the `how` input as `'left'`, we choose to merge the two tables but keep the keys from the left table as the index, dropping the other keys from the right table that do not have a counterpart on the left table.

```
len(csec.index.values)
len(continents.index.values)
merged = pd.merge(left=csec, right=continents, left_index=True,
    right_index=True, how='left')
len(merged.index.values)
merged.head()
```

We now have an extra column named continent at the end of the table.

We need to check for missing values in the new column!

```
missing = merged.Continent.isnull()
missing
```

We can use a `Series` of boo leans as an index to slice another `Series`:

```
merged.Continent[missing]
countries = merged.Continent[missing].index.values
countries
```

Korea, Russia and Slovakia are missing continents.

We can fix those:

```
for country, cont in zip(countries, ['Asia', 'Asia', 'Europe'])
   :
    merged.loc[country, 'Continent'] = cont
merged.Continent
```

Let's reset the index so that `Country` becomes a column again:

```
merged = merged.reset_index()
merged.head()
```

Let's change the index so that we split the data by continent and by country: That is, let's create a `MultiIndex` with `Continent` as the first level, and `Country` as the second level:

```
merged = merged.set_index(keys=['Continent', 'Country'])
merged.head()
merged
```

Let's sort by the `Continent` index so that the table is easier to visualize:

```
merged = merged.sort_index(level='Continent')
merged
```

10 Aggregating Data

Now that we have a nicely formatted panel, we can aggregate and visualize the data.

Time average of the wages by country and by continent:

```
merged.mean(axis=1)
# axis=1: will average across all columns
```

Time series of average wage by Continent:

```
# average across all rows of each Continent
merged.mean(axis=0, level='Continent')
```

Time average of wages by continent:

```
merged.mean(axis=0, level='Continent').mean(axis=1)
```

Highest wage in each continent at each year:

```
merged.max(axis=0, level='Continent')
```

Notice that the columns are times, but there is an issue with the values:

```
merged.columns
```

The data type is object, but we want it to be `datetime`:

```
pd.to_datetime(merged.columns)
merged.columns = pd.to_datetime(merged.columns)
merged.columns.name = 'Time'
merged.columns
```

Having the correct type for dates allows us to easily slice by year:

```
# Highest wage on 2018 by continent
merged['2018-1-1'].max(axis=0, level='Continent')
```

An alternative and more general approach to aggregating data is with the `groupby` method. The idea is to split the table into groups (defined by some criteria), then apply a function to each of the groups (group by group), and then combine the results in a table again.

Let's compute the time average of the wages by country and by continent:

```
grouped = merged.groupby('Continent')
```

`grouped` is a collection of groups. It is basically a dictionary, where each key corresponds to one of the groups defined by the splitting rule. In this case, the rule was to split by Continent, so each key is going to be the name of the continent. We can see each of the groups like so:

```
for key, group in grouped:
    print(f"Key: {key}")
    print(f"Group:\n {group.head()}\n")
```

We can now apply a function to each group to aggregate the values:

```
grouped.mean()
# Equivalents:
merged.groupby('Continent').mean()
merged.mean(axis=0, level='Continent')
```

Another example. Get the number of countries in each group (continent):

```
merged.groupby('Continent').size()
```

We can also supply custom functions to aggregate the data:

```
# Compute median
merged.groupby('Continent').aggregate(np.median)
# Compute variance
merged.groupby('Continent').aggregate(np.var)
# Compute several values at once
merged.groupby('Continent').aggregate([np.mean, np.median, np.
   argmin])
# Can choose to apply functions to specific columns
merged.groupby('Continent').aggregate({'2017-1-1': np.sum})
```

```
# Or even different functions per column
merged.groupby('Continent').aggregate({'2017-1-1': np.sum,
                                       '2018-1-1': np.max})
```

You should read the Group By reference page for further details.

We can use the result of the grouping to plot:

```
merged.transpose()\
      .groupby(level='Continent', axis=1)\
      .mean()\
      .plot(kind='line')
```

Let's create a bar plot of the time average real minimum wage for the different countries:

```
time_avg = merged.transpose().mean(axis=0)
time_avg
time_avg = time_avg.droplevel(0)
time_avg
time_avg.sort_values().plot(kind='bar')
```

11 Resampling

When working with data index by time, we can leverage Pandas functions to re-sample our data to different frequencies. What follows is a simple example, but for a more in depth understanding you should read the Time Series and Date functionality reference page.

Change index to time:

```
merged = merged.transpose()
merged
```

Change frequency to "year start":

```
merged.index.freq = 'AS'
merged.index
```

Pandas identified all dates started in january, so it appended the `JAN` to reflect that. See the Frequency (Offset Aliases) reference page for the possible frequencies.

The `resample` method is similar to a group by, but works with time indices. To down-sample to biyearly, while averaging the values:

```
merged.resample('2AS').mean()
```

Other aggregation methods are available. To down-sample to biyearly, but taking the most recent value instead of the average:

```
merged.resample('2AS').first()
```

We can also up-sample:

```
# Upsample to every month
merged.resample('MS').first()
# Interpolate to fill the missing values
merged.resample('MS').first().interpolate(method='linear')
# Use spline to interpolate
merged.resample('MS').first().interpolate(method='spline',
   order=3)
```

We used the `interpolate` function to fill the missing values when upsampling.

We can plot to see the interpolation results:

```
upsampled = merged.resample('MS').first().interpolate(method='
    spline', order=3)
# Drop Continent and select Brazil
brazil = upsampled.droplevel(0, axis=1)['Brazil']
# Plot the values
ax = brazil.plot(kind='line', color='green', linewidth=0.5)
brazil.resample('YS').first().plot(ax=ax, kind='line', style
    ='.', color='black')
ax.legend(['Interpolation (Cubic Spline)', 'Original Data'])
ax.grid(True, linestyle='--')
```

12 Statsmodels

The Statsmodels package can be used to estimate several classical statistical models. It is built on top of Pandas and Numpy. We will learn how to use its linear regression tools.

Let's start by loading some financial data (Fama-French 3 factors):

```
url = "https://raw.githubusercontent.com/python-for-economists/
    lecture-notes/master/supporting/data/ff3factors.CSV"
factors = pd.read_csv(url, skiprows=3, index_col=0, parse_dates
    =True)
factors
factors.index
factors.index = pd.to_datetime(factors.index, format='%Y%m')
factors.index
```

Let's also load data for the Apple stock:

```
url = "https://raw.githubusercontent.com/python-for-economists/
    lecture-notes/master/supporting/data/monthlyAAPL.csv"
stock = pd.read_csv(url, skiprows=0, index_col=0, parse_dates=
    True)
stock = stock['Close']
stock.index
```

Compute the arithmetic returns:

```
returns = stock.pct_change()
returns
# convert to returns in percentage
returns = 100*returns
returns
```

Let's merge the Fama-French factors with the monthly returns from Apple.

```
merged = pd.merge(left=returns, right=factors, left_index=True,
    right_index=True, how='left')
merged
```

Compute excess stock return:

```
merged['Excess Return'] = merged['Close'] - merged['RF']
merged
```

Drop rows with missing observations:

```
merged = merged.dropna()
merged
```

Let's import Statsmodels:

```
import statsmodels.api as sm
```

The `sm` has a class for OLS. We will run a linear regression of stock return on risk-free rate and market excess return First, we build the model:

```
model = sm.OLS(endog=merged['Excess Return'], exog=merged['Mkt-
   RF'])
type(model)
```

We can use `model` to estimate the parameters:

```
results = model.fit()
type(results)
```

The `results` object holds all the estimates and many other statistics. We can visualize all the information at once by calling the summary method:

```
print(results.summary())
dir(results)
print(results.params)
print(results.rsquared)
```

The `predict` method of `results` can be used to compute $\hat{y}$ for a given x:

```
results.predict(exog=[5.3])
results.predict(exog=[5.3, -2.3, -10.3])
```

We can plot the original data and the estimated model:

```
ax = merged.plot(kind='scatter', x='Mkt-RF',
                 y='Excess Return', alpha=0.4, label='observed
                    ')
ax.scatter(merged['Mkt-RF'], results.predict(),
           color='k', alpha=0.6, label='predicted')
ax.set(title='Predicted Monthly Excess Return from Regression',
       xlabel='Market Excess Return (in %)',
       ylabel='Stock Excess Return (in %)')
ax.legend()
ax.grid(True, linestyle='--', alpha=0.5)
```

Extend the regression with the other factors, one by one

```
model2 = sm.OLS(endog=merged['Excess Return'],
                exog=merged[['Mkt-RF', 'HML']])
model3 = sm.OLS(endog=merged['Excess Return'],
                exog=merged[['Mkt-RF', 'HML', 'SMB']])
results2 = model2.fit()
results3 = model3.fit()
```

To compare the results of several regressions we can use a helper function:

```
from statsmodels.iolib.summary2 import summary_col
```

`summary_col` takes a list of objects representing different OLS results:

```
summary_col([results, results2, results3], float_format='%.2f',
            model_names=('CAPM', 'FF 2 Factors', 'FF 3 Factors
               '),
            stars=True, regressor_order=['Mkt-RF', 'HML', 'SMB
               '])
```

To also display the R-squared and other statistics we need to use the `info=_dict input.` `=info_dict` is a dict, where the keys will be used a labels for the rows and the values are functions that extract the values from the regression results object.

```
info_dict = {'R-squared': lambda r: f"{r.rsquared:.2f}",
             'Total Observations': lambda r: f"{r.nobs:.0f}"}
summary_col([results, results2, results3], float_format='%.2f',
            model_names=('CAPM', 'FF 2 Factors', 'FF 3 Factors
               '),
            stars=True, regressor_order=['Mkt-RF', 'HML', 'SMB
               '],
            info_dict=info_dict)
```

Notice that these summary functions generate tables. You can store the tables in a variable:

```
table = summary_col([results, results2, results3], float_format
   ='%.2f',
            model_names=('CAPM', 'FF 2 Factors', 'FF 3 Factors
               '),
            stars=True, regressor_order=['Mkt-RF', 'HML', 'SMB
               '],
            info_dict=info_dict)
```

And one of the methods of the table is exporting to latex:

```
dir(table)
print(table.as_latex())
```

If you also install the Pyperclip package:

```
pip install pyperclip
```

Then you can automatically copy the table to the clipboard and then you can simply paste it into your latex files:

```
import pyperclip
pyperclip.copy(table.as_latex())
```

Statsmodels is not limited to linear regressions, and also provides functions for estimating:

- Generalized Linear Models
- Generalized Estimating Equations
- Generalized Additive Models

- Robust Linear Models
- Linear Mixed Effects Models
- Regression with Discrete Dependent Variable
- Generalized Linear Mixed Effects Models
- ANOVA
- Time Series
- Vector Autoregression
- Survival and Duration Analysis
- Nonparametric Methods
- Generalized Method of Moments