+Babbel

Coding Challenge for Data Science Candidates at Babbel

General remarks

Data Scientists at Babbel are not developers, and a lot of heavy lifting is done by our great data engineering team. Nonetheless data scientists must be able to code at a more than basic level in order to build robust, reliable and reusable models. Data scientists who are good at programming their stack are simply more effective than those who are not.

At Babbel, Data Science is – at least currently – based on the scientific Python stack (Python 3, Jupyter, NumPy, SciPy, ScikitLearn, PyMC, Pandas, etc.). Therefore, this challenge is focused on your general Python skills, as well as on your mastery of the data science stack.

Some remarks about what we expect:

- The coding challenge consists of three (3) parts. Each part has several tasks. Please do solve all of them.
- We want clean, short, elegant and readable **Python 3** (!) code that makes use of the language's advanced features (such as first class functions, list comprehensions, data structures, etc.) where it makes sense. We neither want BASIC-style Python (doing everything with `if` and `for`), nor code-golf (making things as short and obscure as possible).
- We don't expect production ready code. It is ok to not double-check everything (empty arguments, etc). You can and probably should write tests if it helps you, especially for corner cases, but they are not required, and you don't have to use a formal framework. We look for readability, brevity and elegance. We don't like redundancy.
- All challenges can be solved by writing and calling a bunch of functions. You don't need to and you should not use classes!
- The code you hand in should be runnable on the console / terminal like `python solution_challenge_x.py` … without arguments and without installing external packages with a default Python 3 installation (part 1 and 2)
- Part 3 should be a Jupyter nodenook runnable with an Anaconda 3 distribution.
- For some challenges there are **limitations on what you are allowed to import**! This is documented at the beginning of the respective challenge.
- We sometimes tell you the lines of code of our reference solution for each exercise. This is just a reminder of how short good solutions can be, since many people tend to write to much code. You don't have to match this. None of the tasks requires writing a ton of code!

Part 1 – Basic Python

This challenge must be solved without external packages. This means you may use `import`, but only for things that ship with standard Python. We believe that a single import is enough for a good solution.

You are given a table as a list of dictionaries. Each dictionary represents a row. You may assume that each dictionary has the same set of keys.

Example:

```
table = [   {'age': 32, 'gender': 'm', 'loc': 'Germany', 'val': 4233},
            {'age': 23, 'gender': 'f', 'loc': 'US', 'val': 11223},
            {'age': 31, 'gender': 'f', 'loc': 'France', 'val': 3234},
            {'age': 41, 'gender': 'm', 'loc': 'France', 'val': 2230},
            {'age': 19, 'gender': 'm', 'loc': 'Germany', 'val': 42},
            {'age': 21, 'gender': 'f', 'loc': 'France', 'val': 3315},
            {'age': 23, 'gender': 'm', 'loc': 'Italy', 'val': 520}
        ]
```

Task 1

Write a function

```
group_aggregate(groupby, field, agg, table)
```

which acts like the following SQL statement:

```
SELECT agg f FROM table GROUP BY g
```

Parameters:

`groupby` is a field name by which to group
`field` is a field to aggregate
`agg` is an aggregation function working on iterables
`table` is the table as described above

Output: Print to console as in the examples below.

Example run:

```
group_aggregate(
     groupby = 'gender',
     field = 'transactions',
     agg = sum,
     table = table )
```

```
group_aggregate(
     groupby = 'country',
     field = 'age',
     agg = max,
     table = table )

group_aggregate(
     groupby = 'gender',
     field = 'age',
     agg = min,
     table = table )
```

Console output:

```
aggregating val
m: 7025
f: 17772
aggregating age
Italy: 23
France: 41
US: 23
Germany: 32
aggregating age
m: 19
f: 21
```

Note: Our reference solution has 8 lines (without imports).

Task 2

Write a function `pretty_print(table)` that prints a table nicely, and exactly as follows for the table above:

```
+---+------+-------+-----+
|age|gender|    loc|  val|
+---+------+-------+-----+
| 32|     m|Germany| 4233|
| 23|     f|     US|11223|
| 31|     f| France| 3234|
| 41|     m| France| 2230|
| 19|     m|Germany|   42|
| 21|     f| France| 3315|
| 23|     m|  Italy|  520|
+---+------+-------+-----+
```

- First row is a header with column names.
- Columns ordered alphabetically.
- Columns are right-aligned, and “padded” to maximum content width.

While writing the function you may encounter that if you run your script several times, the order of the columns in your table changes every now and then.

Why is this the case?

Note: Our reference solution has about 20 lines of code, including a few empty ones.

Part 2 – Advanced python

This must be solved <u>without any</u> module imports!

1. Familiarize yourself with generators, if you don't know them. Write a function `take(n,gen)` that returns a list (!) of n elements from the generator `gen`.

2. Familiarize yourself with the `zip` function if you don't know it. Write an infinite generator that zips two iterables of possibly different length, restarting at the beginning of an iterable whenever it is exhausted. The generator shall yield tuples.

   Example:

   ```
   for tup in take( 9, forever_zip("12345", "abc") ):
     print( tup )
   ```

   Output:

   ```
   ('1', 'a')
   ('2', 'b')
   ('3', 'c')
   ('4', 'a')
   ('5', 'b')
   ('1', 'c')
   ('2', 'a')
   ('3', 'b')
   ('4', 'c')
   ```

   Hint: Think of a helper generator that could be useful as a building block.

3. Extend the function to accept an arbitrary number of lists, as in the example.

   Example:

   ```
   for t in take(15,
             forever_zip("12345", "abc", “This is”, “Python”)):
     print t
   ```

   Output:

   ```
   ('1', 'a', 'T', 'P')
   ```

```
('2', 'b', 'h', 'y')
('3', 'c', 'i', 't')
('4', 'a', 's', 'h')
('5', 'b', ' ', 'o')
('1', 'c', 'i', 'n')
('2', 'a', 's', 'P')
('3', 'b', 'T', 'y')
('4', 'c', 'h', 't')
('5', 'a', 'i', 'h')
('1', 'b', 's', 'o')
('2', 'c', ' ', 'n')
('3', 'a', 'i', 'P')
('4', 'b', 's', 'y')
('5', 'c', 'T', 't')
```

Hint: Our solutions for the different tasks have no more than 5 lines of code per function, sometimes significantly less.

Part 3 – Data Science Stack

- Please use the numy / scipy / scikit-learn data science stack for this challenge.
- Please submit a Jupyter notebook.
- Your notebook must be runnable with a default installation of the Anaconda3 distribution.
- If you find you have to make choices or assumptions, describe briefly why you made them.

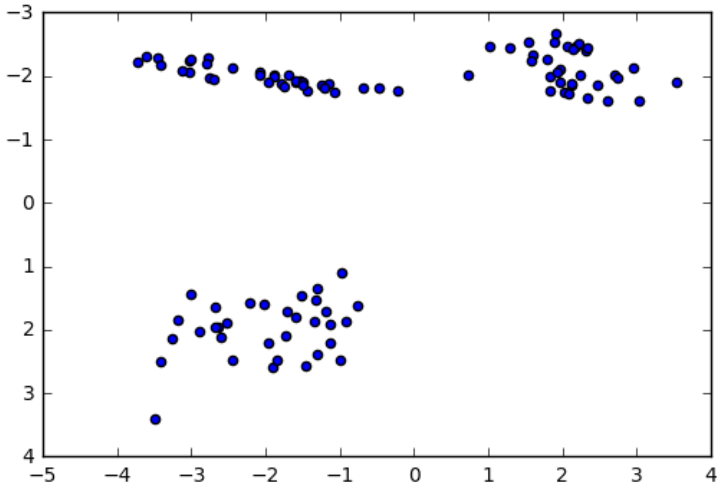
Generate some clustered 2D data as follows:

```
D, _ = sklearn.datasets.make_classification(
            n_samples=200,
            n_features=2,
            n_informative=2,
            n_redundant=0,
            n_classes=3,
            n_clusters_per_class=1,
            class_sep=2 )
```

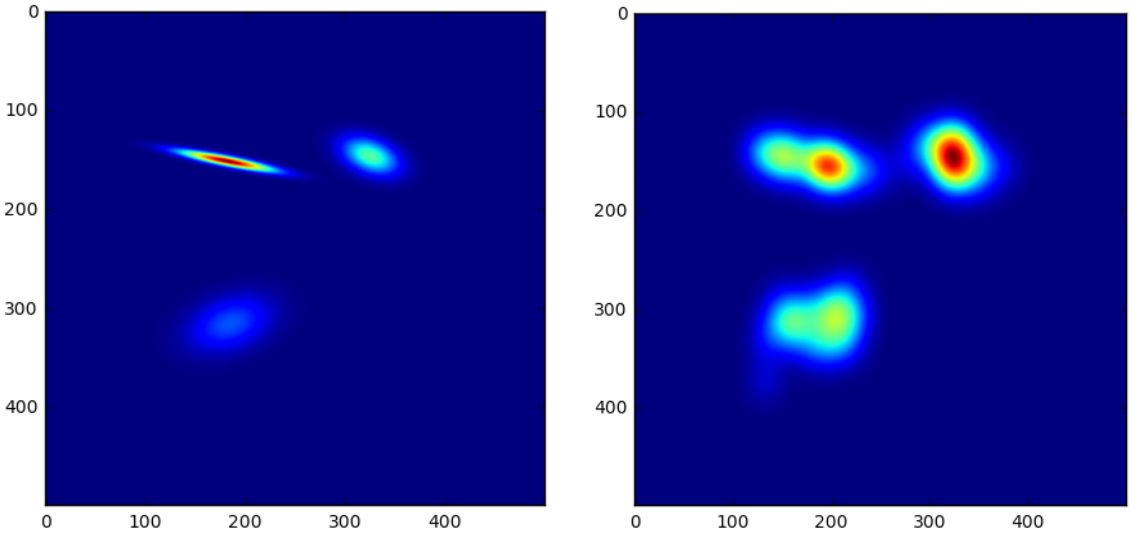
1. Perform density estimation using the following algorithms:
    a. Kernel Density Estimator (KDE) with Gaussian kernel
    b. Gaussian Mixture Model (GMM)
2. Let P(X,Y) denote a 2D distribution you estimated in task 1. Write a function that approximates and plots the marginal distributions P(X) and P(Y), and run it for the densities found by the two algorithms. It may help to write a function that visualizes the 2D densities found by the two methods with a heatmap for checking the plausibility of your marginals.
3. KDE estimates can be noisy if the bandwidth β is too low. Write a function that searches for the smallest β for which the number of modes in the marginal distributions of the KDE estimate (using β) matches the number of modes in the marginal distributions of the GMM estimate.

Examples:

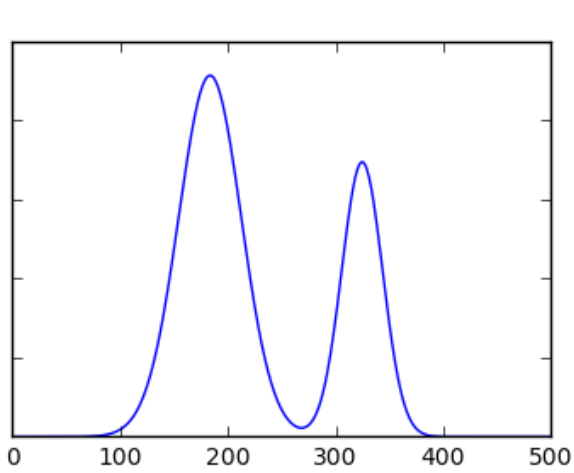
Data points generated by `sklearn.datasets.make_classification`

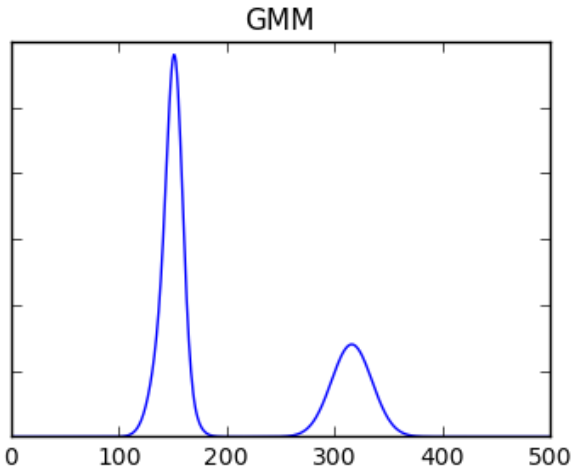


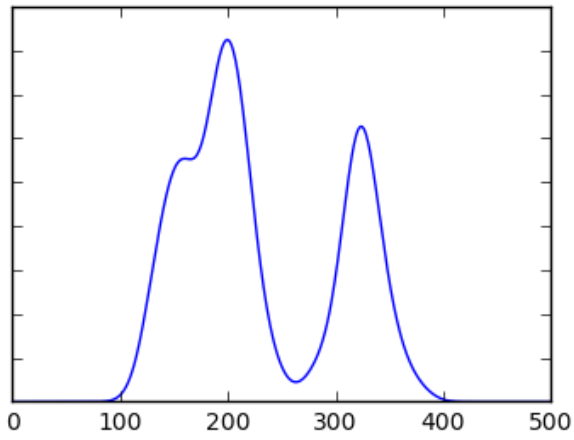
Density heatmaps (GMM and KDE):

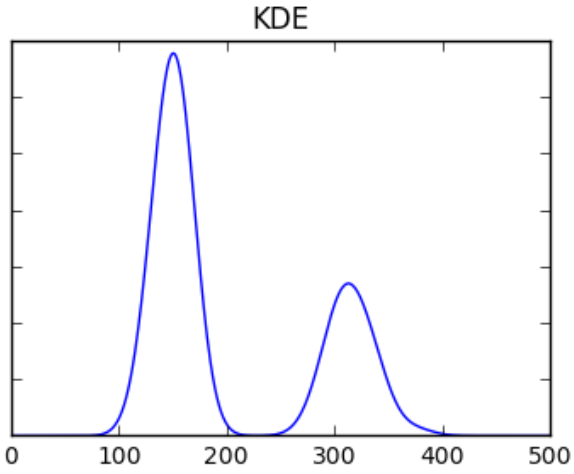


Marginals:


0
100
200
300
400
500


GMM
0
100
200
300
400
500


0
100
200
300
400
500


KDE
0
100
200
300
400
500