Working with Data

Centre for Data Science, ITER
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.

Contents

Introduction

- With any data, our first step should be to explore the data.
- Simplest case is when you have a one-dimensional dataset, which is just a collection of numbers.
- Example: Daily average of no. of minutes a user uses Whatsapp!! $- - >$4 hours or less?
- Next thing one might be interested is in computing few summary statistics. e.g. Smallest, Largest, Mean, Standard Deviation, etc.

Exploring 1-D data

- Suppose you have scores of a batsman, what is the first step you would do to know how well he/she has performed?
- Maybe, it’s going to be mean, but mean sometimes with outliers can give vague ideas.
- A better approach will be to create a histogram and observe it.
- In histogram, we group our data into discrete buckets and count how many points fall into each bucket.
- Let’s create a histogram of a randomly created data.

```python
from collections import Counter, defaultdict
from functools import partial, reduce
from scratch.statistics import correlation,
    standard_deviation, mean
from scratch.probability import inverse_normal_cdf
import math, random, csv
import matplotlib.pyplot as plt
import dateutil.parser
from scratch.probability import inverse_normal_cdf

def bucketize(point, bucket_size):
  """floor the point to the next lower multiple of
      bucket_size"""
  return bucket_size * math.floor(point / bucket_size)

def make_histogram(points, bucket_size):
  """buckets the points and counts how many in each
      bucket"""
  return Counter(bucketize(point, bucket_size) for point in
      points)
```

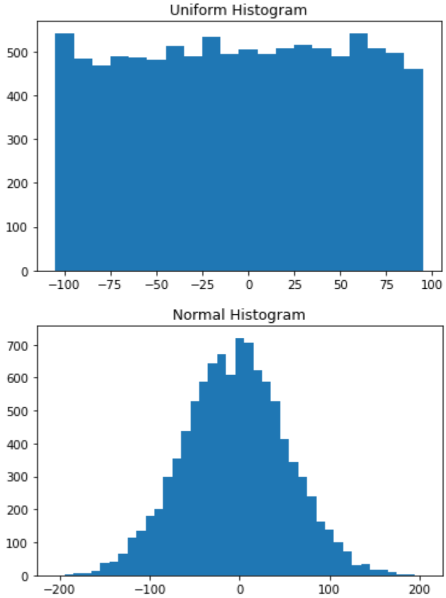
```python
def plot_histogram(points, bucket_size, title=""):
    histogram = make_histogram(points, bucket_size)
    plt.bar(histogram.keys(), histogram.values(),
        width=bucket_size)
    plt.title(title)
    plt.show()

random.seed(0)
uniform=[200*random.random()-100 for _ in range(10000)]
normal=[57*inverse_normal_cdf(random.random()) for _ in
    range(10000)]

plot_histogram(uniform,10,'Uniform Histogram')
plot_histogram(normal,10,'Normal Histogram')
```

Stack operations

1. Means close to 0.
2. Standard deviation of both are close to 58.

However, both are very different distributions.

Results obtained



Exploring 2-D Data

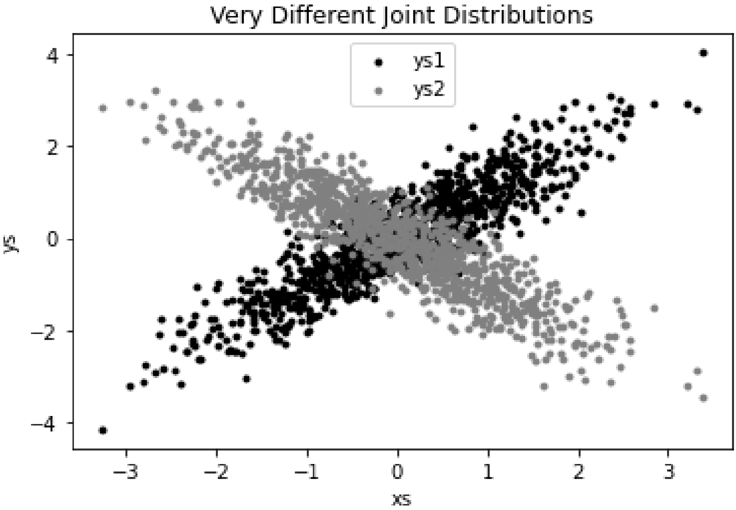
- Now, assume we have one more dimension added to our data.
- Suppose, along with daily minutes, we have years of data science experience of user too.
- We need to understand each dimension individually to have a better idea of data.
- Below is the visualization of a 2-D fake data.

```
from collections import Counter, defaultdict
from functools import partial, reduce
from scratch.statistics import correlation,
    standard_deviation, mean
from scratch.probability import inverse_normal_cdf
import math, random, csv
import matplotlib.pyplot as plt
import dateutil.parser
from scratch.probability import inverse_normal_cdf

def random_normal():
  """returns a random draw from a standard normal
     distribution"""
  return inverse_normal_cdf(random.random())
```

```
xs = [random_normal() for _ in range(1000)]
ys1 = [ x + random_normal() / 2 for x in xs]
ys2 = [-x + random_normal() / 2 for x in xs]

plt.scatter(xs, ys1, marker='.', color='black', label='ys1')
plt.scatter(xs, ys2, marker='.', color='gray', label='ys2')
plt.xlabel('xs')
plt.ylabel('ys')
plt.legend(loc=9)
plt.title("Very Different Joint Distributions")
plt.show()

print(correlation(xs,ys1))
print(correlation(xs,ys2))
```

1. Histogram of both ys1 and ys2 will be similar looking plots.
2. Both are normally distributed with same mean and standard deviation.
3. But they have very different joint distributions.

Results obtained



Many dimensional data

- It can be useful to see how one dimension relates to another.
- This can be seen using a correlation matrix.($(i,j)^{th}$ entry represents relation between i^{th} and j^{th} dimension).
- A more visual approach (if you don't have too many dimensions) is to make a scatterplot matrix showing all the pairwise scatterplots.

Using NamedTuples

- One common way of representing data is using dicts.
- However, accessing things by dict key is error-prone.

```
d={'Closing_price':100}
d['Cosing_price']=200
```

- Instead of showing error or any message, even with this typo code will run.
- There is not helpful way to annotate dictionaries-as-data that have lots of different values, i.e. we can not have hints for these values.
- As an alternative, Python includes a namedtuple class, which is like a tuple but with named slots.
- Like regular tuples, namedtuples are immutable.

```python
import datetime
from collections import namedtuple
StockPrice=namedtuple('StockPrice',['symbol','date',
             'closing_price'])
price=StockPrice('MSFT',datetime.date(2018,12,14),106.03)
assert price.symbol=='MSFT'
assert price.closing_price==106.03
```

```
#Now, we can see how to add hints

import datetime
from collections import namedtuple
from typing import NamedTuple
class StockPrice(NamedTuple):
  symbol: str
  date: datetime.date
  closing_price: float
  def is_high_tech(self) -> bool:
    """It's a class, so we can add methods too"""
    return self.symbol in ['MSFT', 'GOOG', 'FB', 'AMZN',
        'AAPL']
price = StockPrice('MSFT', datetime.date(2018, 12, 14),
    106.03)
assert price.symbol == 'MSFT'
assert price.closing_price == 106.03
assert price.is_high_tech()
```

Now, StockPrice.clo will show suggestion for StockPrice.closing_price

Dataclasses

- Dataclasses are (sort of) a mutable version of NamedTuple.
- The syntax is very similar to NamedTuple.
- We can modify a field of the Dataclass.
- It is available from Python 3.7 or higher versions, lower versions do not support it.

```
import datetime
from dataclasses import dataclass
@dataclass #Decorator
class StockPrice2:
  symbol: str
  date: datetime.date
  closing_price: float
  def is_high_tech(self) -> bool:
    """It's a class, so we can add methods too"""
    return self.symbol in ['MSFT', 'GOOG', 'FB', 'AMZN',
        'AAPL']
price2 = StockPrice2('MSFT', datetime.date(2018, 12, 14),
    106.03)
assert price2.symbol == 'MSFT'
assert price2.closing_price == 106.03
```

```python
#As mentioned, we can modify a dataclass instance's values
# stock split
price2.closing_price /= 2
assert price2.closing_price == 53.015
#Since, it is a regular class, we can also add new fields
price2.cosing_price = 75 #OOPS!!ISSUE
```

So, in this regard NamedTuple is more useful than a dataclass.

Cleaning and Munging

- Mostly, real-world data is dirty.
- We have to convert strings to float(s) or int(s) before we can use them in many processes, such as clustering.
- We have to check for missing values and outliers & bad data.
- Bad data can be something which is not suitable at that place, e.g. having numbers in Name.
- Outliers are which are vague and show that it is either by mistake or by some error is there. e.g. year being 3014 in past data.

Manipulating data

- Most important skills of a data scientist is manipulating data.
- Suppose we have a stock data as:
  data = [StockPrice(symbol='MSFT',date=datetime.date(2018, 12, 24),closing_price=106.03),...]
- We want to know highest-ever closing price for AAPL. Steps should be:

1.Restrict ourselves to AAPL rows.
2.Grab the closing price from each row.
3.Take the max of those prices.

- All three can be done at once as:

```
max_aapl_price = max(stock_price.closing_price
                     for stock_price in data
                     if stock_price.symbol == "AAPL")
```

Manipulating data

- Generally, we might want to know the highest-ever closing price for each stock in our dataset. One way to do this is:
- Create a dict to keep track of highest prices (we'll use a defaultdict that returns minus infinity for missing values, since any price will be greater than that).
- Iterate over our data, updating it.

```
from collections import defaultdict
max_prices: Dict[str, float] = defaultdict(lambda:
    float('-inf'))
for sp in data:
  symbol, closing_price = sp.symbol, sp.closing_price
  if closing_price > max_prices[symbol]:
    max_prices[symbol] = closing_price
```

- Small manipulations are required throughout any data science code.

Rescaling

- Many techniques are sensitive to the scale of the data.
- Suppose we have height(inch) and weight(pounds) as A:(63,150), B(67,170.2) and C(70,177.8).
- If Euclidean distance is computed for this, B's nearest neighbour is A.
- However, if height is converted to cm and then Euclidean distance is calculated, B's nearest neighbour will be C.
- So, if dimensions aren't comparable with each other, we rescale our data so that each dimension has mean 0 and standard deviation 1.
- This gets rid of the units issue.

```
#Below is the code to depict the rescaling of the data
from typing import Tuple
from scratch.linear_algebra import vector_mean
from scratch.statistics import standard_deviation
def scale(data: List[Vector]) -> Tuple[Vector, Vector]:
  """returns the mean and standard deviation for each
      position"""
  dim = len(data[0])
  means = vector_mean(data)
  stdevs = [standard_deviation([vector[i] for vector in
      data])
    for i in range(dim)]
  return means, stdevs
vectors = [[-3, -1, 1], [-1, 0, 1], [1, 1, 1]]
means, stdevs = scale(vectors)
assert means == [-1, 0, 1]
assert stdevs == [2, 1, 0]
```

```python
def rescale(data: List[Vector]) -> List[Vector]:
    """
    Rescales the input data so that each position has
    mean 0 and standard deviation 1. (Leaves a position
    as is if its standard deviation is 0.)
    """
    dim = len(data[0])
    means, stdevs = scale(data)
    # Make a copy of each vector
    rescaled = [v[:] for v in data]
    for v in rescaled:
    for i in range(dim):
    if stdevs[i] > 0:
    v[i] = (v[i] - means[i]) / stdevs[i]
    return rescaled
```

An Aside:tqdm

- Computations can sometimes take a long time.
- In such work, one would like to know that you're making progress and how long you should expect to wait.
- One way of doing this is with the tqdm library, which generates custom progress bars.

```
python -m pip install tqdm
import tqdm
for i in tqdm.tqdm(range(100)):
    # do something slow
    _ = [random.random() for _ in range(1000000)]
```

An Aside:tqdm

- To set description of the progress bar.

```
from typing import List
def primes_up_to(n: int) -> List[int]:
  primes = [2]
  with tqdm.trange(3, n) as t:
    for i in t:
      # i is prime if no smaller prime divides it
      i_is_prime = not any(i % p == 0 for p in primes)
      if i_is_prime:
        primes.append(i)
      t.set_description(f"{len(primes)} primes")
  return primes
my_primes = primes_up_to(100_000)
```

Dimensionality reduction

- Sometimes the “actual” (or useful) dimensions of the data might not correspond to the dimensions we have.
- Most of the variation in the data may seem be along a single dimension that doesn’t correspond to either the x-axis or the y-axis.
- We can use a technique called principal component analysis (PCA) to extract one or more dimensions that capture as much of the variation in the data as possible.
- In practicality the dataset consists of large dimensions and to get a data from all parts(different types) we may require to decrease the dimension as such that the data has most variance.

References

[1] Data Science from Scratch_First Principles with Python by Joel Grus, *O'Reilly*.

Thank You
Any Questions?