Cheatsheet

Libero Biagi

October 8, 2025

Intro

This is a general cheatsheet where I will try to put all the coding, the formulas, the terminology and the possible exercises that might be useful for the exams.

Contents

1 Data Mining

1.1 Theory

1.1.1 Lecture 1

Fernando Becao presents himself and the tutor.
Fernando Becao asks us why we weren't at the party.
We study about supervised and unsupervised learning.
Exam + project.
Exam is 55% of the final grade
Project is 45% and is divided into

- Deliverable 1 -> 30% by November 4th
- Deliverable 2 -> 60% by January 3rd
- Discussion -> 10%

Both exam and project will expire after the year

1.1.2 Lecture 2

Random debate about things
We need to find relevant data to reduce computation and improve the quality of works

Big Data -> we simplify datasets that are too big for normal processing. From big data into mean datasets
Big data characteristics:

- **Volume:** big data are big
- **Velocity:** Big data technology allows databases to process and analyze data while it is being generated
- **Variety:** Big data can be a mixture of structured, unstructured and semi-structured data. To solve this problem Big Data is flexible

Some information about AI, ML and Data Science.

AI: Making things that show human intelligence. Automatize human tasks.

Machine Learning: Approach AI using systems that can find patterns from data and examples. ML systems learn by themselves. We can see them as a sort of subset of AI or a way towards AI

Data Science: the study of where information comes from and how to turn it into a valuable resource.

Data Science vs Data Mining

Data science is a set of principle that guide the extraction of information from data. We try to view problems from a data perspective.

Data mining on the other hand is the extraction of knowledge from data via the use of algorithms.

Find/build attributes is very important. Basically the properties of the things that we are studying have to be selected or preprocessed.

After the construction of the features a ML model can learn how to divide the instances on an hyperplane. This can be used to make predictions. Recall that a model will predict only based on the class that it was trained on.

Is very important to use the relevant and appropriate features. More features can mean more possibility of discriminating between the classes.

Typically we use labeled examples, enough data and clear cut definitions.

If our models try to attribute a label we have supervised learning. Future examples will get a prediction.

To build features we use data warehouse and ETL (extract, transform and load). Recall that we can transform every relational schema into a table (at least with SQL).

Minimum information to identify a costumer:

- Transaction number
- Date and time of transaction
- Item purchased
- Price
- Quantity purchased
- A table that matches product code to its name, subgroup code to name, product group code to group name.
- Product taxonomy to link product code to subgroup code and product subgroup code to product group code.
- Card ID

Consistent behaviors are easier to analyze. To found the level of consistency i can use the standard deviation and I can remove the outliers.

We can also look at relevant variables like:

- Recency
- Frequency
- Monetary value
- Average purchase
- Most frequent store
- Average time between transactions
- Standard deviation of transactional interval
- Costumer stability index
- Relative spend on each product

Canonical tasks in data mining
If we want to classify new data from a decision criterion previously learned we talk about **Supervised learning**

If we want to summarize a data set we talk about **Unsupervised learning**

Supervised learning:

- Classification
- Regression

Unsupervised learning:

- Clustering
- Visualization
- Association

In our datasets the features are the columns while the rows are the instances

Clustering -> We plot the instances on a hyperplane and we try to group them by distance, we can have different plots

Association rules -> based on our data we can use the transactions to find some rule to infer costumer routine. We use confidence, support, lift and so on.

Visualization -> n-D data can be difficult to visualize so we can flatten them into 2-D ones. Examples are histograms, bubbles and goggle boxes. We can also do some dimensionality reduction using PCA or other algorithm.

Data mining process

We can have different methodologies to acquire insights from data.

KDD

1. Data
2. Selection
3. Preprocessing
4. Transformation
5. Data mining
6. Interpretation/Evaluation
7. Knowledge

CRISP-DM

1. Business understanding
2. Data understanding
3. Data preparation
4. Modeling
5. Evaluation
6. Deployment

1.1.3 Lecture 3

We use ML algorithms instead of fixed formulas because the events are too complex for a simple formula.

We don't understand the problems but we try to approximate solution. Black Box ML approach basically.

If the model is not good enough we can either improve the model or gather more data. The second one is quicker. Dumb algorithms with a lot of data will learn better than smart ones with not a lot of them.

Traditional statistics might be described as being characterized by data sets which are small and clean, which are static, which were sampled in an iid manner, which were often collected to answer the particular problem being addressed, and which are solely numeric.

Size of data set is also useful. If we have big datasets even tiny effects exists. They can be useless. Now we should ask if the effect is important or not.

From statistical significance to substantive significance.

Since the datasets are very big we try to compute them in a adaptive or sequential way. Also we might have multiple and interrelated files

We have two main ways to process data

- Incremental

Dimension	Primary data	Secondary data
Definition	Data you collect yourself for a specific, current purpose.	Data collected by others for a different (often past) purpose.
Typical sources	Surveys, experiments, interviews, field measurements, sensors.	Government statistics, research papers, data portals, company databases, syndicated datasets, web-scraped corpora.
Control over design	Full control (sampling, instruments, definitions, timing).	Little/no control; must accept others' design choices.
Fit to your question	High: tailored to your problem and target population.	Varies: often indirect or requires redefinition/derivations.
Cost	Usually higher (money, time, staff, tooling).	Usually lower or free; licensing may apply.
Time to obtain	Longer (planning → collection → cleaning).	Faster (download/access + cleaning/understanding).
Timeliness/recency	Up-to-date by design.	May be outdated; release lags common.
Granularity	Exactly what you need (variables, frequency, detail).	Fixed by source; may lack key variables or be too aggregated.

Table 1: Comparison between primary and secondary data.

- Batch

Other characteristics of problems in data mining are

- Nonstationarity and population drift
- Selection bias
- Spurious relationships

Input variables should be causally related to the output. If we have a small number of observations we will have high correlation.

Confounding variables will correlate both with dependent and independent variables. The confounding factor will estimate incorrectly the relation. A **Spurious relationship** happens when we perceive a relationship between two variables that actually doesn't exist. We ar not accounting for the confounding factor.

Input variables must be causally related to the output to be meaningful.

We have to discriminate between **causality** and **correlation**:

- Correlation -> two things co-occurs, changing one of them will not change the output
- Causality -> a change on the input will cause a change on the output

Correlation is a pattern, causation a consequence

Input Space -> is the input feature vector, the algorithm will look for a solution

Curse of dimensionality -> bad effects can be caused by redundant features, bigger dimensionality means bigger and more sparse input. Clustering can be harder. Generalization is exponentially harder. Feature selection can be an answer.

Input space coverage -> representative training examples will improve our model quality. Test data outside the training input space will be bad for the performance.

Interpolation -> predictions in the range of data that we have
Extrapolation predictions outside the range of data that we have, Time series
Separation -> if we plot our data on a 2-D space we can be able to draw by hand the regions where the data are. ML models will try to do it mathematically. If we can use a linear hyperplane we have **Linearly separable data** if not they are not linearly separable. With separable classes we can get 0 errors. We want to minimize the error (finding the Bayes error). Simple algorithm like perceptron will solve problems with separables classes.

Kind of variables

- Nominal
- Ordinal
- Discrete

- Continuous
- Interval
- Ratio

Metadata -> Information that provides information about data

- Descriptive metadata
- Structural metadata
- Administrative metadata

1.1.4 Lecture 4

Today visualization Strong points of good visualization

- Can show processes
- Show comparisons between numbers
- Can show differences and changes
- Can use geography to show local data
- Can show relationships
- Can show density

What not to do

- Don't clog the graph
- No 3-D
- Don't use unfaithful charts

Guidelines

- Reduce chartjunk
- Increase data-ink ration
- Use similar structures in the same charts

Tufte lie factor -> Measure of distortion in a graph

$$\text{Lie Factor} = \frac{\text{Size of effect in graph}}{\text{Size of effect in data}} \tag{1}$$

As rule of thumb we should have $0.95 < \text{Lie Factor} < 1.05$

Suggestions

- White background
- Don't use colors if not necessary, Charts should be like man suits (Prof, 2025)
- Order the items in a smart way
- Use a scale
- Crisp borders
- No smooth line
- Simple is better
- No 3-D
- Discriminate clearly the clusters with shapes and colors
- Spaghetti chart should highlight interesting patterns
- Clutterplot should highlight interesting points
- Use shadows to highlights interesting points

Charts that can be useful

- Bar charts, vertical and horizontal
- Line charts
- Mix of the previous
- Stacked bar charts
- Scatterplot, also with tendency lines, grids, bubbles
- Pie charts, can mix them with histograms, we can label them, compare with others (same as paired column chart). They are like stacked bar charts. We also have part to whole mini pie charts. We can plot them on a graph.
- Radar plot, easy to compare more of them

Visualization for analysis

- Histogram, can be stacked and combined with scatterplots
- Boxplot, can be combined with scatterplots, histograms

Correlation matrices

- Can be done with distributions, scatter plots and matrices

Parallel coordinate

- Show patterns that can be compared easily

Small Multiples

- Can show many graphs together, easy for comparisons
- Basically we can have a lot of variables in a 2-D graph

Heat maps

- Can show quantities in the plane that we are visualizing

Tree maps

- We can see quantities with sizes and dividing them points based on certain characteristics

Geo-visualization

- We can aggregate data from different geographical point of view, we can use normal maps, bubbles and plot quantities of them. Cartograms are useful for quantities on and densities.

Linked Views

- We can put different vies of the dataset together. We can select different subsets to compare them

1.1.5 Lecture 5

Today data preparation and pre-processing

Preparation:

1. Missing values
2. Outliers
3. Discretization and encoding
4. Imbalanced datasets

Pre processing
wirte roadmap

With data pre-processing and preparation we want to make our data useful for the model. All the datasets are made of signal and noise. We want to maximize the signal and reduce as much as possible the noise.

Real data usually are:

- Incomplete
- Noisy
- Inconsistent

Treating missing data
Missing values are values that are not available, very common.
How to deal:

- Delete records with missing data, biased
- Delete columns with too many missing data, loose information
- Imputing with central tendency metrics, for general and for subsets
- Derive them in an objective way
- Use a predictive model
- Use similarity measures

2 Programming

2.1 Theory and code examples

2.1.1 Lecture 1

Printing →A string is returned and shown on the screen

```
    print("Hello World")
```

print() syntax →print(object(s), separator=separator, end=end, file=file, flush=flush)

Errors →They happen when there is a problem in the code

- **Runtime Errors** →Something wrong, the code won't run
- **Semantic Errors** →The output is not what we expected

We have many more of them, to solve we have to **debug the code**.

Using Jupyter notebooks we can use some special commands that are preceded by the % character

- **%timeit** →determines the execution time of a single-line statement. Performs multiple runs to achieve robust results.
- **%%timeit** →same as %timeit but for the entire cell
- **%time**→determines the execution time of a single-line statement. Performs a single run!
- **%%time**→same as %time but for the entire cell
- **%run**→Run the named file inside IPython as a program
- **%history**→displays the command history. Use %history -n to display last n-commands with line numbers.
- **%recall<line_no>**→re-executes command at line_no. You can also specify range of line numbers
- **%who**→Shows list of all variables defined within the current notebook
- **%Ismagic**→Shows a list of magic commands
- **%magic**→Quick and simple list of all available magic functions with detailed descriptions
- **%quickref**→List of common magic commands and their descriptions

And many more

2.1.2 Lecture 2

Semantics →in Python tabs and spaces are used to structure the code. If you use a colon you have to indent the code in the right way. Semicolons instead can be used for multiple statements on the same line

Objects →everything in Python is an object with it's own methods, functions and characteristics

Comments →# denotes comments

To represent information we can use different kind of data types and structures

Variables →a variable can take whatever value we want

```
    a = 2 #variable declaration
    print(a)
>>> 2
    type(a)
>>>int
```

Other possible types are

- float
- string
- complex
- boolean

If we want to save a collection of objects we can use

```
List = [1, 3, "a", 7]
Tuple = (1, "a", 8, 7)
Dictionary = {"a":1, "b":4}
Set = {1, "a", 4, 8}
```

Data Structure	Mutable / Immutable	Ordered / Unordered	Indexed / History
List	Mutable	Ordered	Indexed
Dictionary	Mutable	Unordered	History of addition
Tuple	Immutable	Ordered	Indexed
Set	Mutable	Unordered	No indexing (unique elements)

Table 2: Comparison of Python collection data structures.

Every data structure has it's own methods.
When we are using indexed data structures we have to remember that the first index is 0.

Since we want to work with data and modify them we need a way to do it. Operators can take variables and make computations if those are compatible

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Table 3: Python arithmetic operators with names and examples.

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Table 4: Python assignment operators with examples.

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Table 5: Python comparison operators.

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverses the result, returns False if the result is True	not(x < 5 and x < 10)

Table 6: Python logical operators.

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Table 7: Python bitwise operators.

Operator	Description
`is`	Returns True if both variables are the same object
`is not`	Returns True if both variables are not the same object

Table 8: Python identity operators.

Flow control →to check that our program is doing what we want we can apply different statements.
if →we define a condition under which a certain action is done if that condition is reached

Operator	Name	Example
`==`	Equal	`x == y`
`!=`	Not equal	`x != y`
`>`	Greater than	`x > y`
`<`	Less than	`x < y`
`>=`	Greater than or equal to	`x >= y`
`<=`	Less than or equal to	`x <= y`

Table 9: Python comparison operators with examples for conditional statements.

elif →it means else if, we specify a new condition
else →to be put as last, if neither if or elif are met we use else

```
if name == "Libero":
    print("Hello")
elif name == "Josè":
    print("Forza Milan")
else:
    print("None")
```

Loops →for now we just cared about one time operations, we can do those on multiple times with loops. We have two main loops

- **for loops** →we specify an interval and the action is performed that number of times.
- **while loops** →the operation is made as long as a certain condition is true

```
for x in range (10):
    print(x)

a = 0
while a < 25:
    print(a)
    a +=1
```

Some problems can arise if the condition of the while loop is never met.

Comprehension →for faster performances we can use this particular structure. It works for all the mutable data structure

```
    squares = [x**2 for x in range(10)]
    print(squares)

    even_squares = [x**2 for x in range(10) if x % 2 == 0]
    print(even_squares)


    squares_dict = {x: x**2 for x in range(5)}
    print(squares_dict)

    squares_set = {x**2 for x in range(5)}
    print(squares_set)
```

With this general structure

```
    {key_expression: value_expression for item in iterable if condition}

    {expression for item in iterable if condition}

```

Slices →if we want a subset of the data structure.

General structure

```
    list[start:stop:step]
```

Negative indexes mean to start from the end

typecast →we can transform a variable with a certain structure into another with another structure

```
    # Typecasting examples in one snippet

# String to int
x_str = "10"
x_int = int(x_str)
print(x_int, type(x_int))  # 10 <class 'int'>

# String to float
y_str = "3.14"
y_float = float(y_str)
print(y_float, type(y_float))  # 3.14 <class 'float'>

# Int to string
z_int = 100
z_str = str(z_int)
print(z_str, type(z_str))  # '100' <class 'str'>

# Tuple to list
tup = (1, 2, 3)
lst = list(tup)
print(lst, type(lst))  # [1, 2, 3] <class 'list'>

# List to tuple
lst2 = [4, 5, 6]
tup2 = tuple(lst2)
print(tup2, type(tup2))  # (4, 5, 6) <class 'tuple'>

# Int to float
```

```
num = 7
num_float = float(num)
print(num_float, type(num_float))  # 7.0 <class 'float'>

# Float to int
num2 = 9.8
num2_int = int(num2)  # truncates decimal part
print(num2_int, type(num2_int))  # 9 <class 'int'>

```

2.1.3 Lecture 3

Functions →like a mathematical function a python one will take some inputs and will apply the same operations to get a certain result. We define them to avoid writing the same code over and over.

```
    def function(input): #to define
        input operation
        return output

    function(other_input) #to call
```

A function can also work with the elements of a list or other collections of elements using a for loop.

If we want to have a flexible number of inputs we can use *args and **kwargs

- ***args** →the function will take a list as input and will do the computations on each element of that list
- ****kwargs** →same as args but with a list of keys and values, the function will return a dictionary

```
    def multiply(*args):
        output = 1
        for n in args:
            output *= n
        return output

    print(multiply(2, 3, 4))

```

```
    def my_function(**kwargs):
        for key, value in kwargs.items():
            print(key, value)

    my_function(name="Alice", age=25, city="Paris")

    ###Output
name Alice
age 25
city Paris
```

Link for a video if it's still confusing

You can add infos about your function with the ___doc___

```
    def greets(*args):
        """This function says Hello"""
        for name in args:
            print("Hello", name)
    greet.__doc__
```

If we want a fast singular use function we can use **lambda functions**.

```
lambda arguments: expression

add = lambda x, y : x + y
add(2, 3)
```

Map →we want to transform certain values into others in a fast way

```
list(map(lambda x : x * 2, [1, 2, 3, 4]))
```

Filter →if we want to select a subset of our data

```
from random import sample

X = sample(range(-25, 25), 25)
f = filter(lambda x: x > 0, X)
```

When we want to call a certain function multiple times we can use two main ways:

- **Iteration** →We define a number of times and the function will do its job for that number.
- **Recursion** →The function will call itself until it reaches a base case. It's important to define base, edge and general cases. Useful when we work with trees or graphs

```
def factorial(n):
if n == 1:
    return 1   #base case
elif n == 0:
    return 1   #edge case
else:
    return n * factorial (n-1) #general case
```

We should differentiate between:

- **Functions** →defined by def or lambda, they can be applied to almost everything if well defined.
- **Methods** →associated with certain objects
- **Attributes** →variables associated to certain objects

Remember kids, if you want to code something probably someone has already done it. So why bother? We can import the work done by other people

```
import module as mod
from module import method1, method2

```

sys.path will give us all the directories that our interpreter is watching
Useful modules can be:

- csv
- datetime

- io
- json
- math
- os
- random
- sqlite3
- xml
- zipfile
- zlib

Namespace A namespace is a collection of names that reference objects.
- **Built-in Names**: predefined names available in every Python interpreter. Examples: `list`, `dict`, `map`, `tuple`.

```
import builtins
print(dir(builtins))   # list built-in names
```

- **Global Names**: user-defined names created in the main program body (variables, functions, classes).

```
x = 42   # global variable
def foo():
    return x
print(globals())   # list global names
```

- **Local Names**: names defined inside a function, valid only inside it.

```
def bar():
    y = 10  # local variable
    print(locals())   # list local names
bar()
# print(y)  -> Error: y does not exist in the global scope
```

Scope Scope defines where a variable can be accessed.
- **Global Scope**: variable accessible throughout the whole program. - **Local Scope**: variable accessible only inside the function where it was declared.

```
animal = "dog"  # global variable

def test_scope():
    # local variable with the same name
    animal = "cat"
    print("Local:", animal)

test_scope()
print("Global:", animal)
```

Output:

```
Local: cat
Global: dog
```

Using the keyword global It allows modifying a global variable inside a function.

```
count = 0

def increment():
    global count
    count += 1

increment()
print(count)  # 1
```

2.1.4 Lecture 4

Today we start with Pandas, fav library for data.

```
    import pandas a pd #always import as pd
```

Series →a Pandas series is 1-D object that can hold any data type. Is made of 2 arrays, one for the index and the other one with the actual data.

```
    obj = pd.Series([11, 28,72, , 5, 8])

    obj.array #information on the array
    obj.index #information on the indices
```

The left column is the index one, always. We can use custom index lists

```
    fruits = ['apples', 'oranges', 'cherries', 'pears']
    quantities = [20, 33, 52, 10]
    S = pd.Series(quantities, index=fruits)
    #the index list is fruits

    S2 = pd.Series([17, 13, 31, 32], index=fruits)
    print(S + S2) #will sum the elements on the same position
    sum(S) #will tell us the total of the nubers in the array

    fruits2 = ['raspberries', 'oranges', 'cherries', 'pears']
    S2 = pd.Series([17, 13, 31, 32], index=fruits2)
    print(S + S2) #the operations are aligned by index, elements that don't appear in both lists
    ↪  will be NaN, like a Joint in relational algebra

    print(S["apples]) #we can access like a dictionary, output will be 20
    S["apples"] = 25 #to modify the element

    print(S + 2) #to add 2 to every element
    print(np.sin(S)) #to apply the sin function

    S.apply(lambda x: x if x > 25 else -1 * x) #apply will make a function to all the elements of
    ↪  the series, Map() will work element wise

    print(S[S > 25]) #to filter using booleans

    "apples" in S #to see if a key exists

    cities = {"London": 8615246,
    "Sesto San Giovanni": 79121,
    "Montevideo": 1405798}
    city_series = pd.Series(cities) #from dictionary to series

    print(cities.isnull()) #to see which element is NaN
    print(cities.notnull()) #to see which element is not Nan
    print(cities.fillna(0)) #to change Nan to 0.0 (floats)
```

```
print(cities.fillna(0).astype(int)) #to change Nan to 0 (integers)

obj.name = "Libero" #to give our series a name
obj.index.name = "diocane" #to give the index comlumn a name
obj.index #we see a list of the index column
```

Characteristics of Index object

- Immutable
- Behaves like a fixed size set (we can check if a certain index is in the column with the in method)
- Can contain duplicate labels

We have several useful methods for the index objects.

If we put multiple series one after the other we get a matrix called **DataFrame** which can be considered like an Excel spreadsheet. Like Excel a pandas dataframe has both row and column index.

```
pd.concatenate([row1, row2, row3]) #to connect the rows if they share the same index
pd.concatenate([row1, row2, row3], axis=1) #they become columns, default is 0

df.colum.values #retrieve the columns names
df = pd.Dataframe(df, index= a_list) #to rename the index column with the names of a list
df = pd.Dataframe(df,
columns=("name2", "name3", "name1") #to rearrange the columns order, same as
↪  df.reindex(["name3", "name1", "name2"])

df.rename(columns={
"name1" = "newname1",
"name2" = "newname"
}, inplace=True) #to rename columns, inplace False will return a copy of the dataset

df = pd.Dataframe(df, columns=["name1", "name2"], index=df["other_column"]) #to put
↪  other_column as new index

df.loc("label", "other_label") #select all the rows with that label
df.loc(df.condition > number) #for conditions

df.sum() #sum on all the columns
df["column1"].sum() #sum on a single column
df["column1"].cumsum() #cumulative sum

#to add columns we put the new column name in the column list and we impute it with
↪  df["new_col"] = df["old_col].cumsum() for example. Default values for new columns are NaN

df["attribute"] #access a column same as df.attribute()

df["attribute"] = number #will fill the column with that number, for unique values use a list
↪  of the same size

df.T #to get the transpose

df = pd.read_csv("path/to/csv/file")

df.head(n) #to see the first n lines of the dataframe

df.to_csv("path/where/you/want/to/save/the/file")

df.loc[] #access with label based approach, KeyError if it can't find the value
df.iloc[] #access with index, IndexError if the requested index is out of bounds, except for
↪  slice indexers

df["b": "m"] #slices can also work by labels, we will take also the middle values

```

How to select subsets

- Fetching like a dictionary, select the index that you want
- Slice by index
- Slice by labels
- Fetch multiple entries
- Condition based selection

You should use .loc[] and by index

iloc vs loc

- iloc = by label
- loc = by index
- duloc = lord Farquad city

```
    df.loc["label"] #select a row by index label
    df.loc["label", ["col1", "col3"]] #row and column selection
    df.loc[:"label", ["col1", "col3"]] #as before but with slicing

    df.iloc[[2, 1]] #fetch rows with that position
    df.iloc[[1, 2], [3, 0, 1]] select certain rows and columns
    df.iloc[:, :3] #with slices
```

2.1.5 Lecture 5

To see statistical characteristics of our dataset we can do

```
    df.describe() #the arguments are percentiles, include and exclude
```

To see how many null values and the object in the column

```
    df.info() #the arguments are verbose, buf, max_cols, memory_usage, null_counts
```

To see the number of unique values

```
    df["name_col"].value_counts() #the arguments are normalize, sort, bins, dropna
```

To group or divide kind of data, basically binning

```
    df["col1"] = df["col1"].map(lambda x: operation on x)
    df.groupby("col1") ["col2"].value_counts() #with parameters by, group_keys, dropn, as_index
```

If we want to make a multi dimensional visualization of group by we can use pivot

```


pd.pivot(index="col1", columns="col2", values="col3") #with arguments data, values, index,
↪  columns, aggfunc

```

To aggregate df using a relational algebra join

```
df3 = pd.merge(df1, df2) #with arguments right, how, on, left_on, right_on
```

To make queries (SQL style)

```
df.head() #to see first 5 rows
df.query("price < 150").head() #with argument inplace.

#we can apply and, or and so on, basically SQL queries inside a string
```

For 1-Hot encoding

```
pd.get_dummies(df["key"], dtype=int) #to join with another df that has those columns we use

df = df[["data"]].join(dummies)
```