The Fundamentals of MongoDB Aggregation

RDB

Databases are spreadsheets with metadata : For patients' work email, entire new entry would be made
In RDB, data is organized with Primary keys (as index) in different tables : So, to look up at single patient, we traverse dozens of databases tables

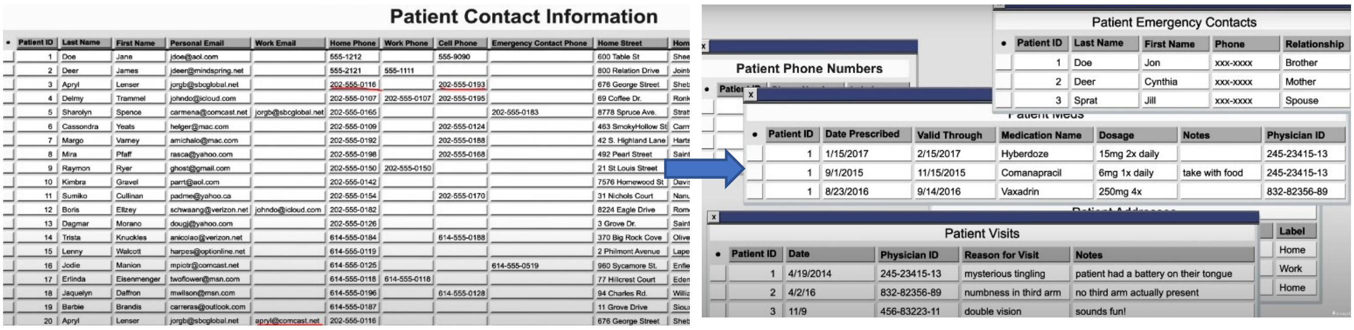


Figure 1 Realtional databases : complication

- This complicates data understanding ability to write applications
- Hard to add features
- Fetching data from many sources [tables] is inefficient

MongoDB

Document Model

Structuring with flexible schema is:

- Natural for programmers to read and code
- Easy for computers to process

Redundancy - Scalability - cloud native - **Fundamentally Distributed**

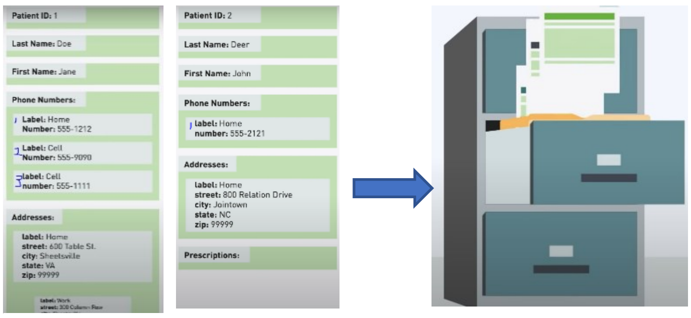


Figure 2 MongoDB document structure : flexible schema

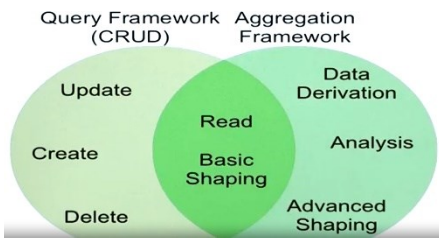
MongoDB Query vs MongoDB Aggregation

Aggregation framework enables developers to define FUNCTIONAL PIPELINES for data Preparation – Shaping – Analytics. Fundamental components are-

- **Stages** : Grouping-Sorting-Shaping etc.
- **Expressions** : Logic units of functionality

Why we use aggregation?

- Need to prepare data
  - Cleaning
  - Deriving information
- Ask preliminary questions



Concept of Functional Pipeline

Assembly line of 1:N stages configurable for transformations using 1:N aggregation operators Or Expressions

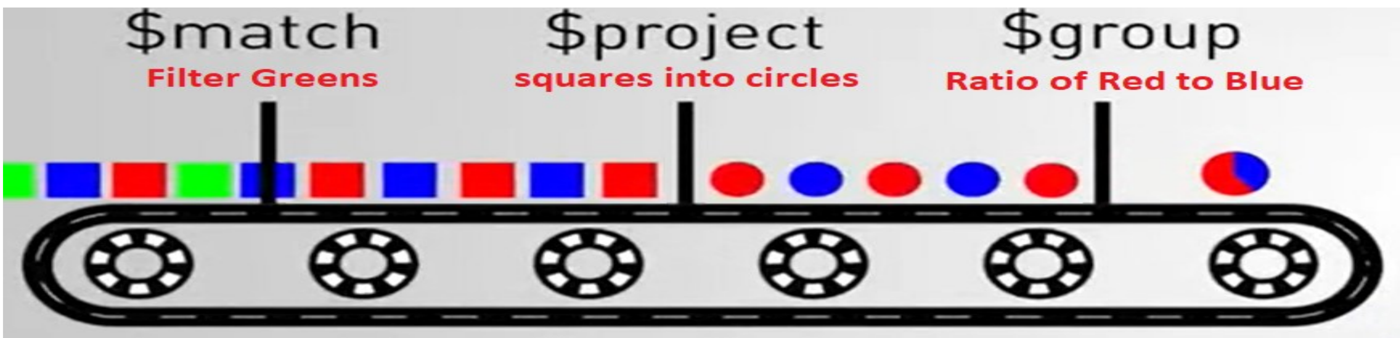


Figure 3 Projection pipeline

Aggregation framework: Structure and syntax

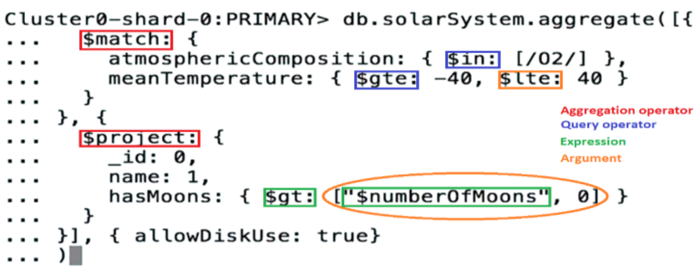
[Aggregation Pipeline Quick Reference]()

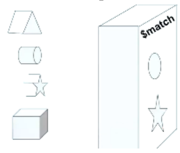
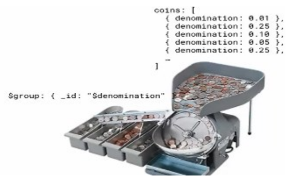
- **Operators** typically appear in **Key** position
- **Expressions [act like functions]** typically appear In **Value** position

```
Field Path:  "$fieldName"    ("$numberOfMoons")
```
Access field in doc

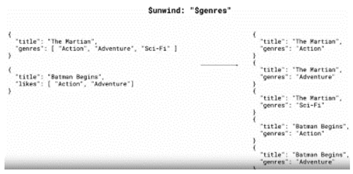
```
System Variable:  "$$UPPERCASE"  ("$$CURRENT")
```
System level variable 'CURRENT'

```
User Variable:  "$$foo"
```
User variable



Operator	Syntax	Description	Details
MATCH	db.<DBname>.aggregate([{ $match : {<query>} }])	• Cannot use $where • Does not have projection • Match should be first operator if $text is used • First stage match increases query throughput as it can take adv of Indexing	**Filter operation** 
PROJECT	db.<DBname>.aggregate([{ $project : {_id:0, <specs>} }])	• Select, Remove, Reassign, derive new fields • Similar to **"map"** function in python • Mention all fields to retain; others are removed auto • Except "_id" : needs explicit removal	
GROUP	db.<DBname>.aggregate([{ $group: {_id: <expression>, <field1>: { <acc1>: <expr1> },...} }])	• Group by a column naturally • Layer of detail using accumulator such as "total_amt_coin_type" : {$sum : 1}	

REDUCE	db.<DBname>.aggregate([{
	$reduce: {
	input: <array>,
	initialValue: <expr>,
	in: <expr> }
	}])
		• Act on "input" array with "initial" value [acc] with logic "in"; and update "init" at each stage	

UNWIND	db.<DBname>.aggregate([{
	$unwind: {
	path: <field path>,
	includeArrayIndex: <string>,
	preserveNullAndEmptyArrays: <Bool>
	}}])
		• unwind an array field creating a new document for every entry where each field values are now a separate entry.	

MAP	db.<DBname>.aggregate([{
	$map: {
	input: <expression>,
	as: <string>,
	in: <expression> }
	}])
		• Applies an expression to each item in an array and returns an array with the applied results.	

LET	db.<DBname>.aggregate([{
	$let: {
	vars: { <var1>: <expression>, ... },
	in: <expression>}
	}])
		• Binds variables **"var"** for use in the specified expression and returns the result of the **"in"** expression.	

LOOKUP	db.<DBname>.aggregate([{
	$lookup: {
	from: <collection to join>,
	localField: <fields input documents>,
	foreignField:<field "from" collection>,
	as: <output array field> }
	}])
		• **Left outer join** – strict equality comparison
		• **Collection to "from" CAN NOT be sharded**	

GRAPH LOOKUP	db.<DBname>.aggregate([{
	$graphLookup: {
	from: <collection>,
	startWith: <expression>,
	connectFromField: <string>,
	connectToField: <string>,
	as: <string>,
	maxDepth: <number>,
	depthField: <string>,
	restrictSearchWithMatch: <document>}
	}])
		• Run on primary shard only in sharded environment
		• **Collection to "from" CAN NOT be sharded**
		• **Final 100MB RAM limit – No effect {allowDiskUsage : True}**	

EXPRESSIVE LOOKUP	db.<DBname>.aggregate([{
	$lookup: {
	from: <collection to join>,
	let: { <var_1>: <expression>, …, <var_n>: <expression> },
	pipeline: [<pipeline to execute on collection to join>],
	as: <output array field>
	}
	}])
		• Regular lookup is essentially join – creates new field DISSASOCIATED WITH OLD SUBFIELDS
		• Not good for a NoSQL db : hence old subfields are typically remapped
		• In expressive lookup, Shaping is done before joining : same task within less RAM
		• Pipelines execute with context of collection specified in "from" field	

ADD FIELDS	db.<DBname>.aggregate([{
	$addFields : {
	<newField>: <expression>,
	... }
	}])
		• appends new fields to existing documents	

FACET	db.<DBname>.aggregate([{
	$facet:
	{
	<outputField1>: [<stg1>, <stg2>,..],
	<outputField2>: [<stg1>, <stg2>,..],
	...
	}
	}])
		• categorize and group incoming documents	

SET DIFFERENCE	db.<DBname>.aggregate([{
	{ $setDifference:
	[<expression1>, <expression2>]
	}
	}])
		• Input sets [A,B] and yields values that exist only in A	

ARRAY ELEMENT AT	db.<DBname>.aggregate([{ $arrayElemAt: [<array>, <idx>] }])	• Returns the element at the specified array index	

SET UNION	db.<DBname>.aggregate([{ $setUnion: [<expression1>, <expression2>,...] }])	• Takes two or more arrays and returns an array containing the elements that appear in any input array	
SAMPLE	db.<DBname>.aggregate([{ $sample: { size: <positive integer> } }])	• Randomly selects the specified number of documents from its input	
EXPR	{ $expr: { <expression> } }	• Allows the use of aggregation expressions within the query language.	
FIRST, LAST	{ $first: < expr > } { $last: < expr > }	• only for **GROUP** • Returns the value that results from applying expression to first/ last document in a group sharing _id	
MULTIPLY	db.<DBname>.aggregate([{ $multiply : [<expr1>, <expr2>,<expr3>, ...] }])	• Expr1*Expr2*…ExprN	
DIVIDE	db.<DBname>.aggregate([{ $divide: [<expr1>,<expr2>] }])	• Expr1 / Expr2	
SUM , MAX, MIN, Average, Standard Deviation	{ $sum: <expression> } { $max: < expr > } { $min: < expr > } { $avg: < expr > } { $stdDevPop: <expr> } { $stdDevSamp: <expr> }	• Have memory within scope of document supplied within project. Need unwind and group to operate on all documents	

MongoDB expressions

Boolean
$and
$or
$not

Set
$setEquals
$setIntersection
$setUnion
$setDifference
$setIsSubset
$anyElementTrue
$allElementsTrue

Comparison
$cmp
$eq
$gt
$gte
$lt
$lte
$ne

Variable
$let

Literal
$literal

Arithmetic
$abs
$add
$ceil
$divide
$exp
$floor
$ln
$log
$log10
$mod
$multiply
$pow
$sqrt
$subtract
$trunc

String
$concat
$indexOfBytes
$indexOfCP
$split
$strLenBytes
$strLenCP
$strcasecmp
$substrBytes
$substrCP
$toLower
$toUpper

Text Search
$meta

Array
$arrayElemAt
$arrayToObject
$concatArrays
$filter
$in
$indexOfArray
$isArray
$map
$objectToArray
$range
$reduce
$reverseArray
$size
$slice
$zip

Data Type
$type

Conditional
$cond
$ifNull
$switch

Date
$dayOfYear
$dayOfMonth
$dayOfWeek
$year
$month
$week
$hour
$minute
$second
$millisecond
$dateToString
$isoDayOfWeek
$isoWeek
$isoWeekYear

Accumulators
$sum
$avg
$first
$last
$max
$min
$push
$addToSet
$stdDevPop
$stdDevSamp

Cursor like stages

- { $**sort**: { <field1>: <sort order>,..} : sort on particular field/s

- Use early in pipeline to take advantage of indexes. If put after project stage, it performs in-memory sort
- Sort has default 100MB RAM limit – to exceed, use aggregation option {allowDiskUsage : True}

- { $**skip**: <positive integer> } : return all skipping first N → [N:]
- { $**limit**: <positive integer> }: return first N → operation [:N]
- { $**count**: <string – name of output field> }: return count of documents returned

Links:
[Query and Projection Operators]

Leveraging MongoDB’s Flexible Schema

Importance of Schema Design

Operator	Syntax	Description	Details
MONGO IMPORT	mongo "mongodb+srv://cluster0.jotdp.mongodb.net/<dbname>" --username rasikMooc Mongoimport --uri mongodb+srv://rasikMooc:rasikMooc@cluster0.jotdp.mongodb.net/aggregation-framework-mooc --collection retail --type csv --file retail.csv --headerline --drop	• Import csv/ uri data • --drop for removing earlier data • --mode= [upsert/insert/merge] -- upsertFields=[key fields] • **No shape change / derived fields**	
CREATE VIEW	db.createView(<view>, <source>, <pipeline>, <options>)	•	
SYSTEM VIEW	db.system.views	• contains information about each view in the database	

ADD TO SET	{
	$addToSet: {
	<field1>: <value1>, ... } }
		• Add values to array omitting duplicates	

PUSH	{
	$push: {
	<field1>: <value1>, ... } }
		• Appends a specified value to an array	

Schema Exploration

Make schema explicit

- In SQL, schema is about DATA & RELATIONS
- In NoSQL, schema is about **Performance**
  - Access patterns and frequent queries : What is purpose behind query
- **Schema validation is optional**

Exploration options

- Collection.find_one() : naïve
- mongoDB Compass

Data Migration Processes

- Mongoimport [import, change field type] and aggregation [shape change, field derivation] together offer fetch and process capacity
- In-place operations cannot be done with aggregation

VIEWs

db.createView(<view>, <source>, <pipeline>, <options>)

- Views are public
- Non materialized – computed separately for each read operation
- “Aggregation Pipeline as a collection”
- Create slices of DB
  - **Horizontal** [eg. $match - select reduced number of documents (based on criteria) with **same shape** i.e. cardinality]
  - **Vertical** [eg. $project – return same no of documents with **different shape** i.e. reduced cardinality (based on criteria)]

- **No permission for : write / mapReduce/ index/ $text/ $geoNear/ renaming/ find() with $elemMatch, $slice, $meta**

Supplementary schema with accumulators : $ addFields

Use cases of $graphLookup

Tree like architectures

Entity resolution

Regular Lookup

```
MongoDB Enterprise > db.air_alliances.aggregate([
...   {
...     $lookup: {
...       from: "air_airlines",
...       localField: "airlines",
...       foreignField: "name",
...       as: "airlines"
...     }
...   },
...   {
...     $addFields: {
...       airlines: {
...         $map: {
...           input: "$airlines",
...           in: {
...             name: "$$this.name",
...             alias: "$$this.alias",
...             id: "$$this.airline"
...           }
...         }
```

expressive lookup

```
MongoDB Enterprise > db.air_alliances.aggregate([
...   {
...     $lookup: {
...       from: "air_airlines",
...       let: { constituents: "$airlines" },
...       pipeline: [
...         {
...           $match: { $expr: { $in: ["$name", "$$constituents"] } }
...         },
...         {
...           $project: {
...             _id: 0,
...             name: 1,
...             alias: 1,
...             id: "$airline"
...           }
...         }
...       ],
...       as: "airlines"
...     }
...   }
... ]
```

giving same outcome

```
"_id" : ObjectId("5980bef9a39d0ba3c650ae9d"),
"name" : "OneWorld",
"airlines" : [
        {
                "name" : "Canadian Airlines",
                "alias" : "CP",
                "id" : 1615
        },
        {
                "name" : "British Airways",
                "alias" : "BA",
                "id" : 1355
        },
        {
                "name" : "American Airlines",
                "alias" : "AA",
                "id" : 24
        },
```

Machine Learning with MongoDB

Pearson-Rho [Pearson correlation coefficient OR bivariate coefficient]
- Strength of linear relation between variables

$$r_{xy} = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n\sum x_i^2 - (\sum x_i)^2}\,\sqrt{n\sum y_i^2 - (\sum y_i)^2}}$$

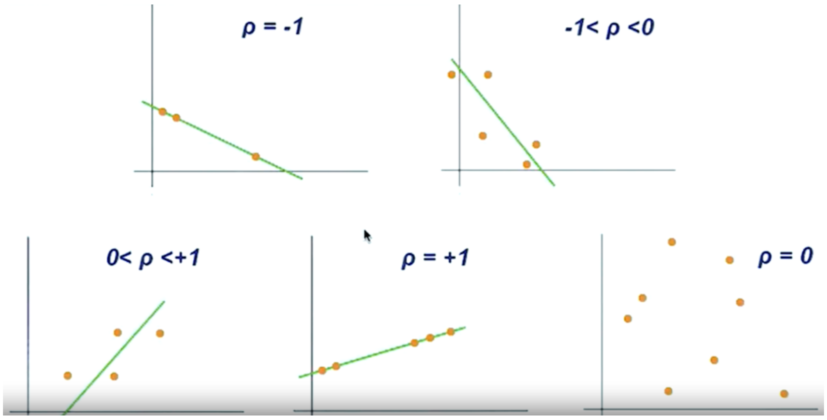


Figure 4 Perason's Coefficient of correlation

Market Basket Analysis

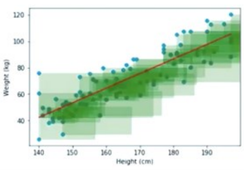
Total Inventory I = {apple, beverage, chips…]
Bought Basket b = subset{I}

Associative rules
- **Support : likelihood of a basket**
  - Support(b) : N(b) / N(Transaction)
- **Confidence : likelihood of a basket[A,B] given an item[A]**
  - Confidence(A→B) : Support(U{A,B}) / Support(A)
  - Drawback : if consequent[i.e. B] is popular; then implication is indicated, though there isn't any.
- **Lift : Likelihood of another item[B] given an item[A]**
  - lift(A→B) : Support(U{A,B}) / [Support(A)* Support(B)]
  - lift = 1 NO association
  - lift < 1 - ve association
  - lift > 1 + ve association

Principal Component Analysis - Reduce noise by reducing dimensionality
Mathematical steps for PCA:
- Input multidimension data
- Drop target variable
- Calculate covariance matrix of [independent variables]^T
- Calculate eigenvalue and corresponding eigenvectors
- Sort by eigenvalue
- Take cumulative % sum of eigenvalue
- Choose top N eigenvalues contributing towards n%
- Use eigenvectors of chosen eigenvalues as new Dimensions for further regression/ classification/ clustering

Linear Regression
- Least square & Mean squared error : minimize area under the squares

MongoDB help transforming large data and pipe it into scikit-learn



Linear Regression Methods

- Ordinary least squares
- Generalized least squares
- Percentage least squares
- Iteratively reweighted least squares
- Total least squares
- Maximum likelihood estimation
- Ridge regression
- Least absolute deviations
- Bayesian linear regression
- Quantile regression
- Principal component regression
- Least-angle regression

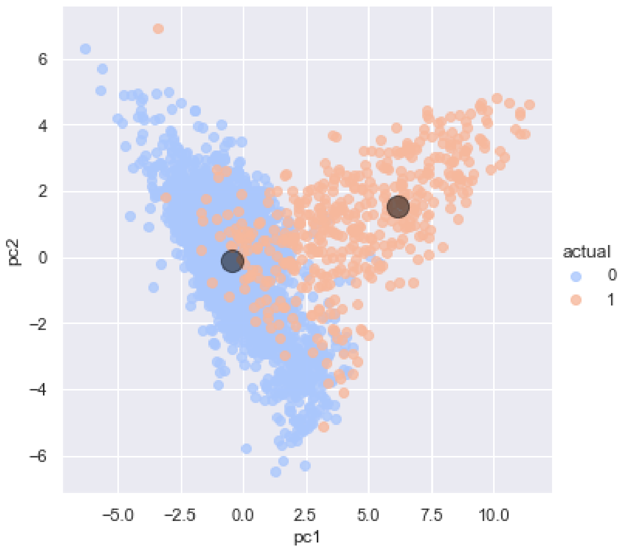
Decision Tree
Precision : True +ve vs false +ve [imprison most notorious ones, but might miss few less important ones]
Recall : True + against false -ve [general model : imprison all rather than missing few]
F1 score : harmonic mean of Precision and recall
Random forest : Ensemble of decision trees [averages results from all models]

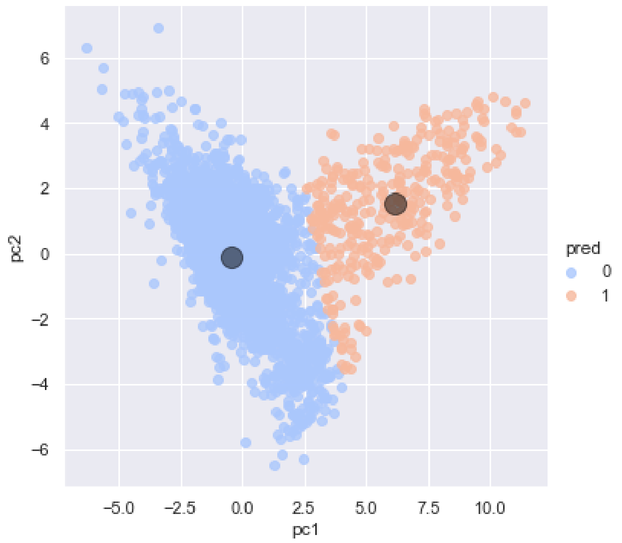
Clustering

- Assign random centroids
- Calculate d{point, centroid}
- Assign point to nearest centroid [dmin]
- Tune centroid to minimize distance from all points in its cluster
- Repeat

- **k-Means clustering**

Data normalization helps improving accuracy





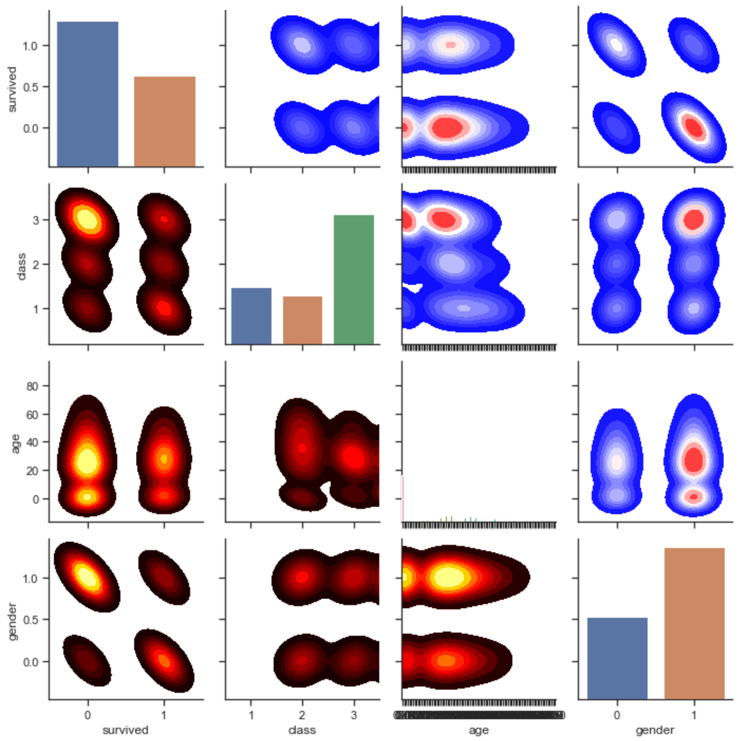


Figure 5 Titanic Dataset correaltion distribution