ANOMALY DETECTION

Nick Radcliffe
Stochastic Solutions Limited
& Department of Mathematics, University of Edinburgh

PyData London Workshop
27 April 2018

RESOURCES

```
Numpy:                      pip install numpy
Pandas:                     pip install pandas
TDDA library:               pip install tdda
Feather format:             pip install feather-format
Enhancements for
  feather format:           pip install pmmif

Material for workshop (inc slides):

git clone \
    https://github.com/tdda/pydatalondon2018ad.git

Docs: tdda library: http://tdda.readthedocs.io

TDDA generally:     http://tdda.info
```

PART I: CONCEPTS

WHAT IS ANOMALY DETECTION?

Anomaly: Deviation from “normal”/“expected”

- Intrinsically dependent on expectation/definition of normality

Examples:

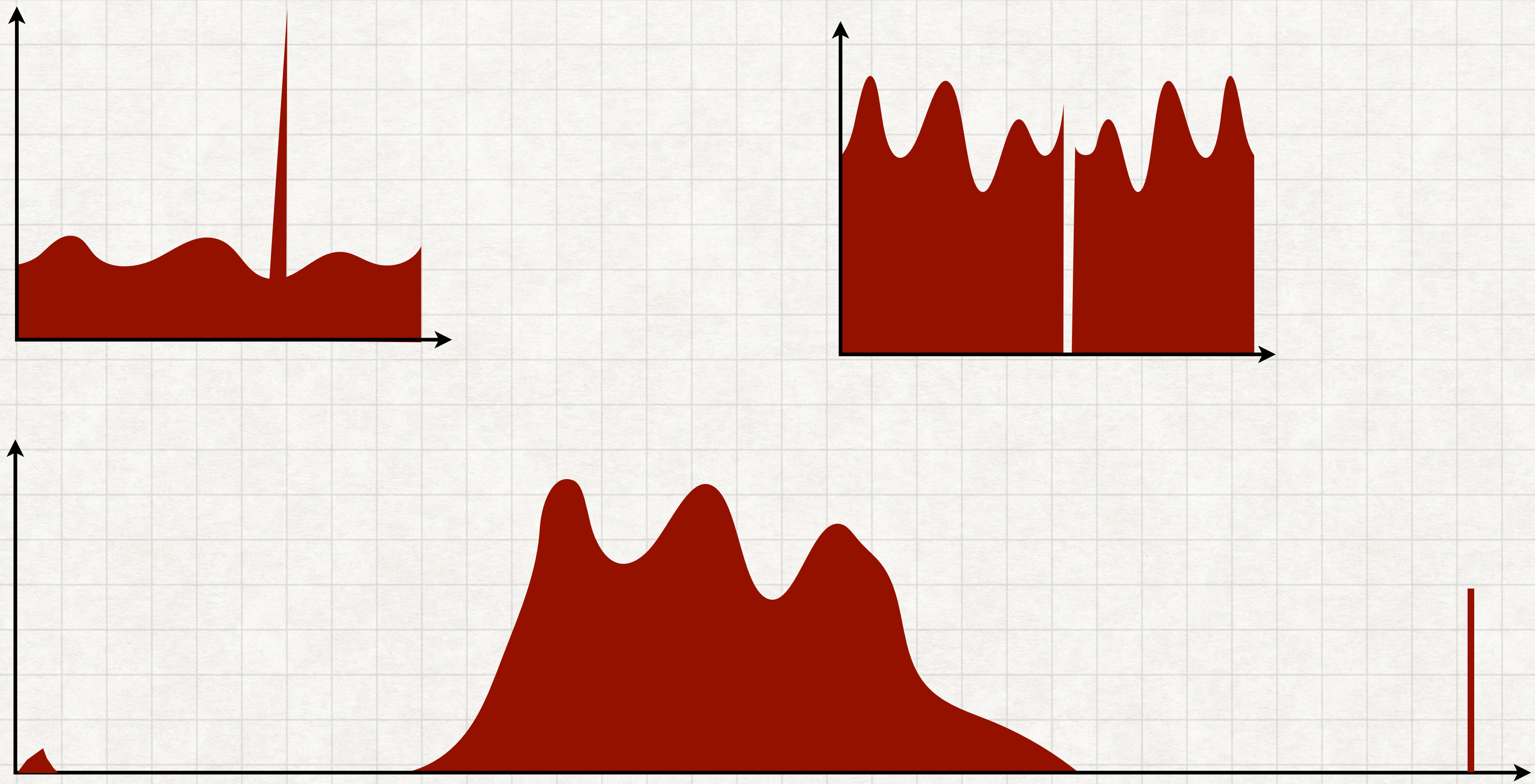
- Detect problems, e.g. fraudulent credit card transactions, systems failures (e.g. server down), gaming (e.g. bogus ecommerce ratings), medical problems (e.g. heart arhythmia).
- Also, detect opportunities: buying patterns suggesting new markets/offers, low usage periods suggesting savings, identifying unusually successful staff/methods/approaches etc.

AUTOMATED ANOMALY DETECTION

- Outlying values in one dimension/variable
- Outlying combinations of values
- Patterns of regularity or irregularity in data streams
- Areas of high or low density
- Illegal data values
- Repetition

WHAT ANOMALIES LOOK LIKE (CONCEPTUALLY)

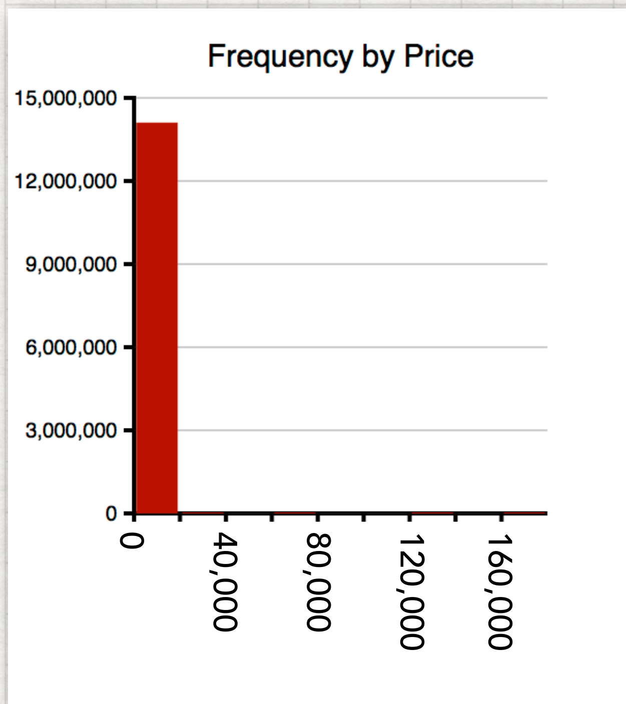
Outlying values in One Dimension



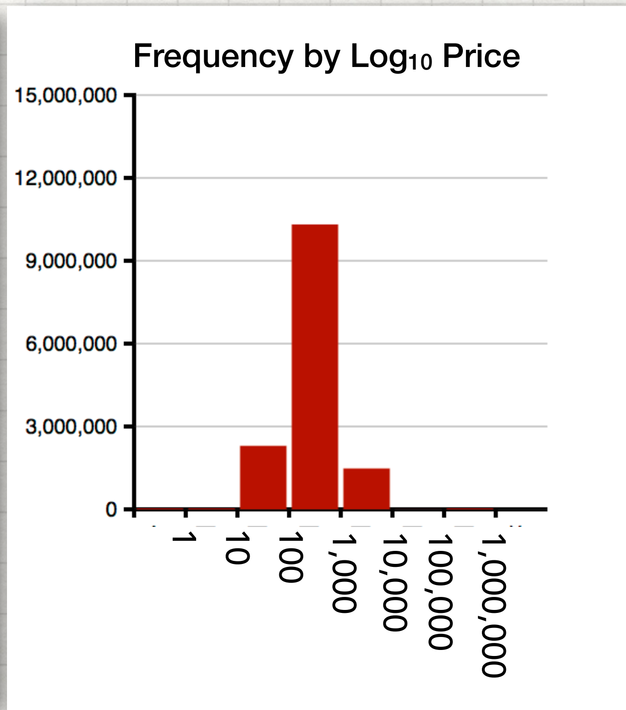
Often much more extreme than shown here, making plotting tricky

WHAT ANOMALIES OFTEN LOOK LIKE IN PRACTICE

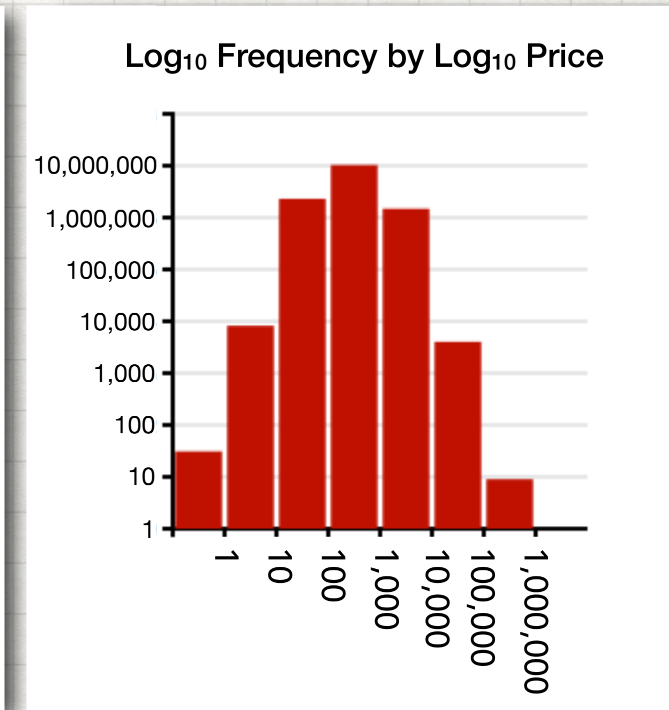
Using Logarithmic or Log-Log Plots to Display Outliers can help



Linear Histogram



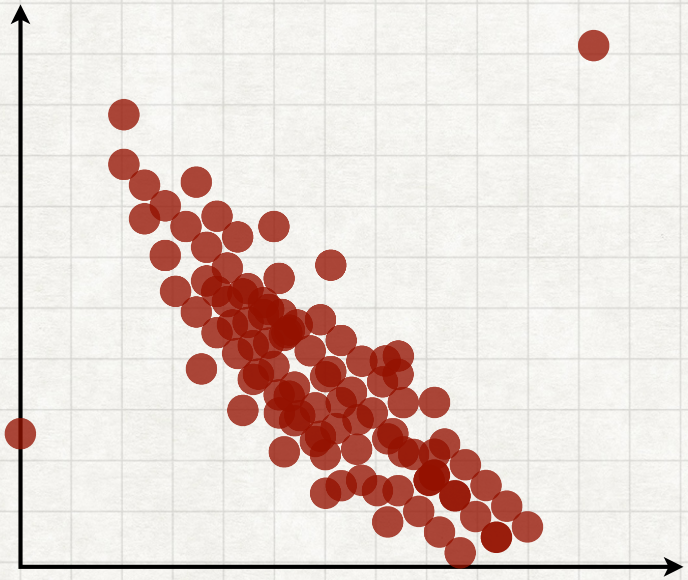
Histogram
Frequency vs. Log_{10} Price

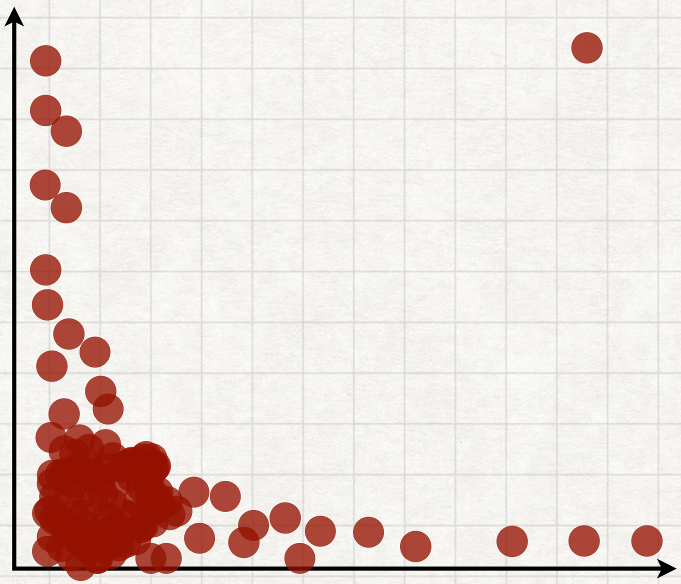


"Log-Log" Histogram
Log_{10} Freq vs. Log_{10} Price

WHAT ANOMALIES LOOK LIKE?

Outlying combinations of values

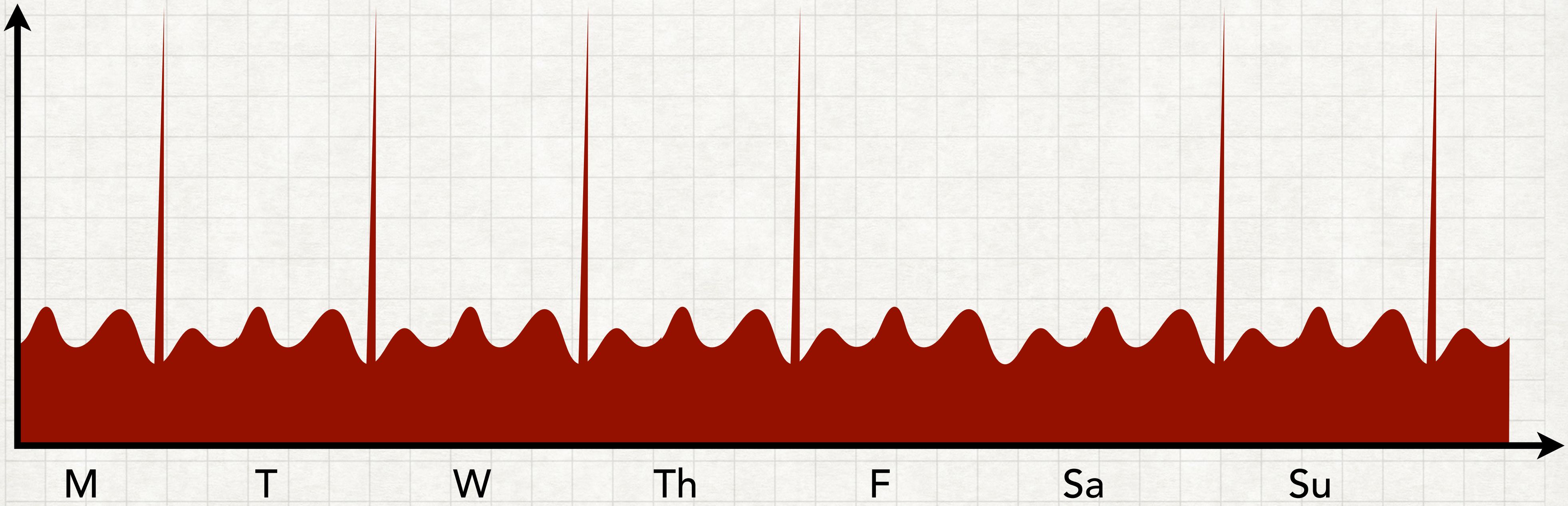




Again, may need log scales or similar to see in practice

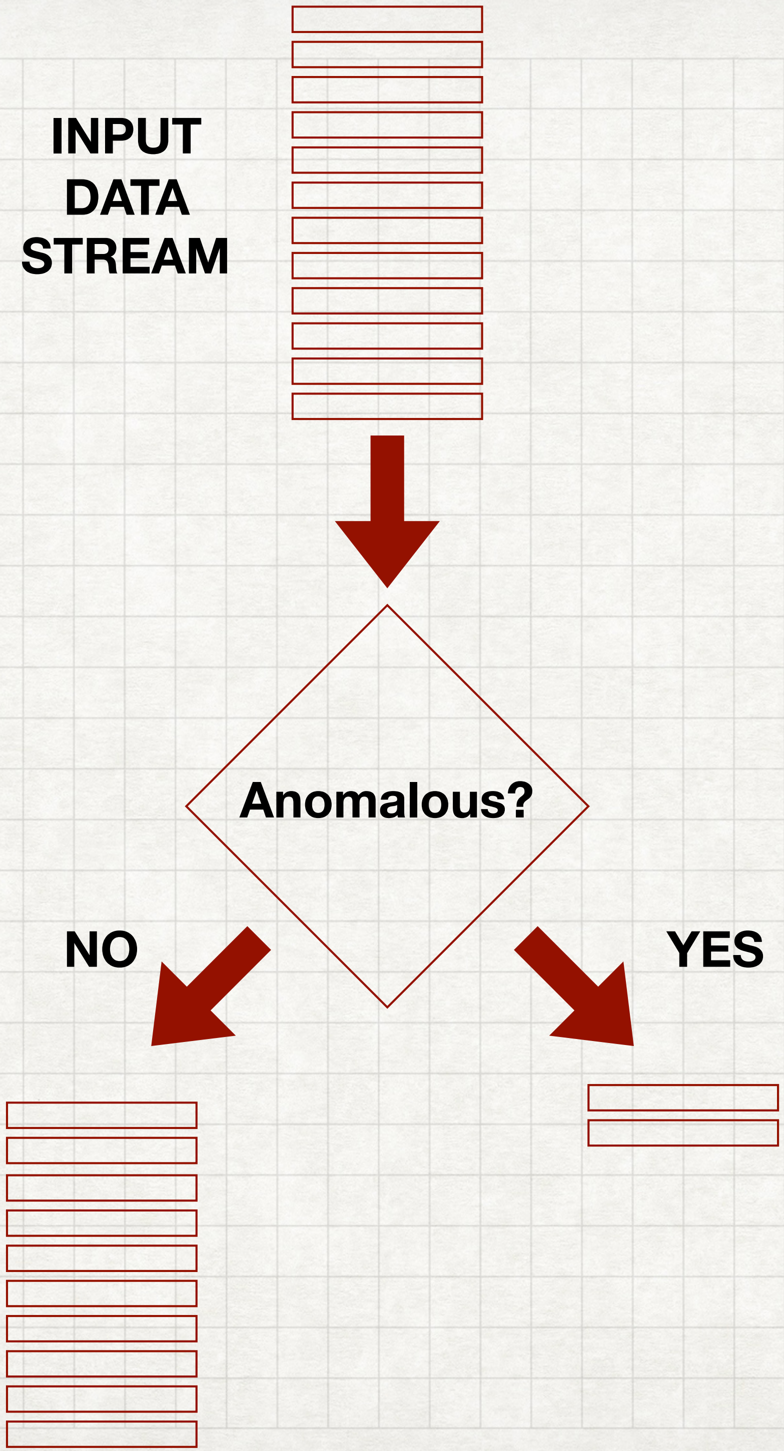
WHAT ANOMALIES LOOK LIKE?

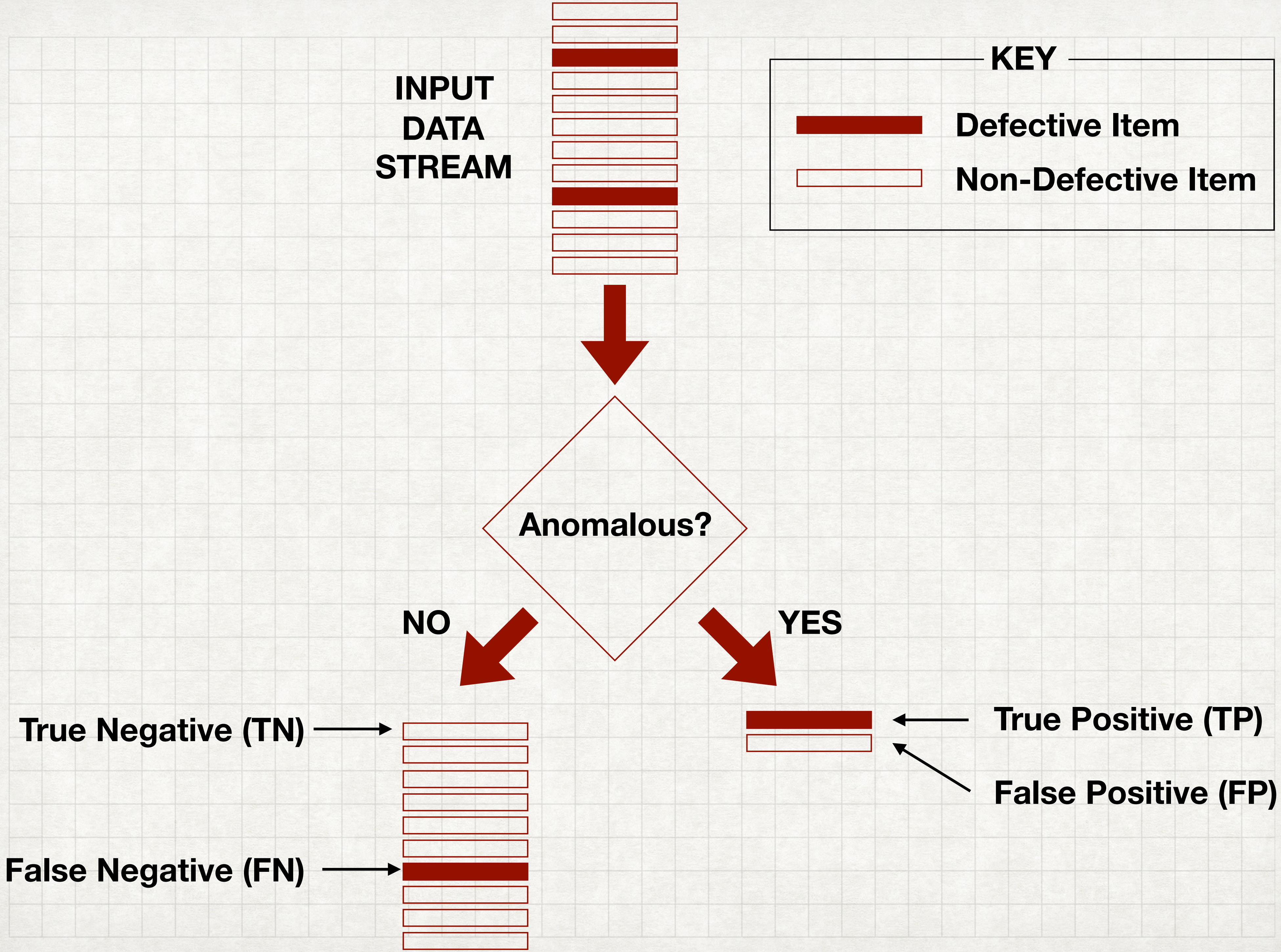
Regularity



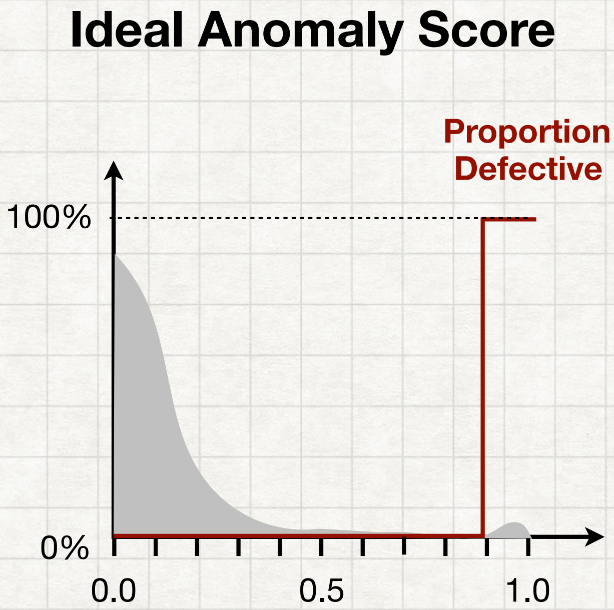
Are the spikes anomalous?
If so, are the anomalous because of their size, their regularity or both?
Is the missing spike anomalous?
Could it be a backup process that runs 6 days a week?

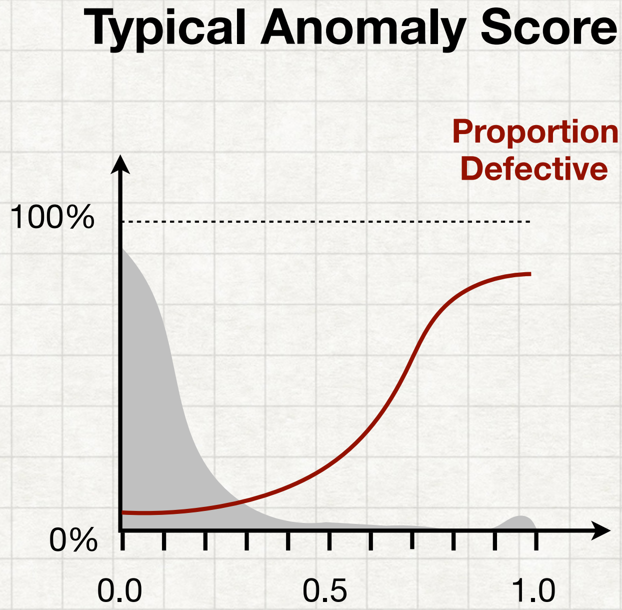
AUTOMATED ANOMALY DETECTION




INPUT
DATA
STREAM
KEY
Defective Item
Non-Defective Item
Anomalous?
NO
YES
True Negative (TN)
False Negative (FN)
True Positive (TP)
False Positive (FP)

ANOMALY SCORES





FALSE POSITIVE & FALSE NEGATIVE RATES

False Positive Rate — *Number of False Positives* — *Number of True* ***Negatives***

$$FPR = FP \,/\, (FP + TN)$$

the proportion of non-defective items that are flagged as anomalous

False Negative Rate — *Number of False Negatives* — *Number of True* ***Positives***

$$FNR = FN \,/\, (FN + TP)$$

the proportion of defective items that are flagged as non-anomalous

FALSE POSITIVE & FALSE NEGATIVE RATES

EXAMPLE

	detector flags as anomalous
10 000 items total	
1% defective	80%
99% non-defective	10%

$$FPR = FP \,/\, (FP + TN)$$

$$FNR = FN \,/\, (FN + TP)$$

TP	FN
FP	TN

EXERCISE: FILL IN

EXERCISE: Confirm formula gives FPR of 10% and FNR of 20%

EXERCISE: Calculate proportion of items flagged by the detector that are actually defective

FALSE POSITIVE & FALSE NEGATIVE RATES

EXAMPLE

10 000 items total	detector flags as anomalous
1% defective	80%
99% non-defective	10%

$$FPR = FP / (FP + TN)$$

$$FNR = FN / (FN + TP)$$

TP	FN
80	20
FP	TN
990	8910

EXERCISE: FILL IN

EXERCISE: Confirm formula gives FPR of 10% and FNR of 20%

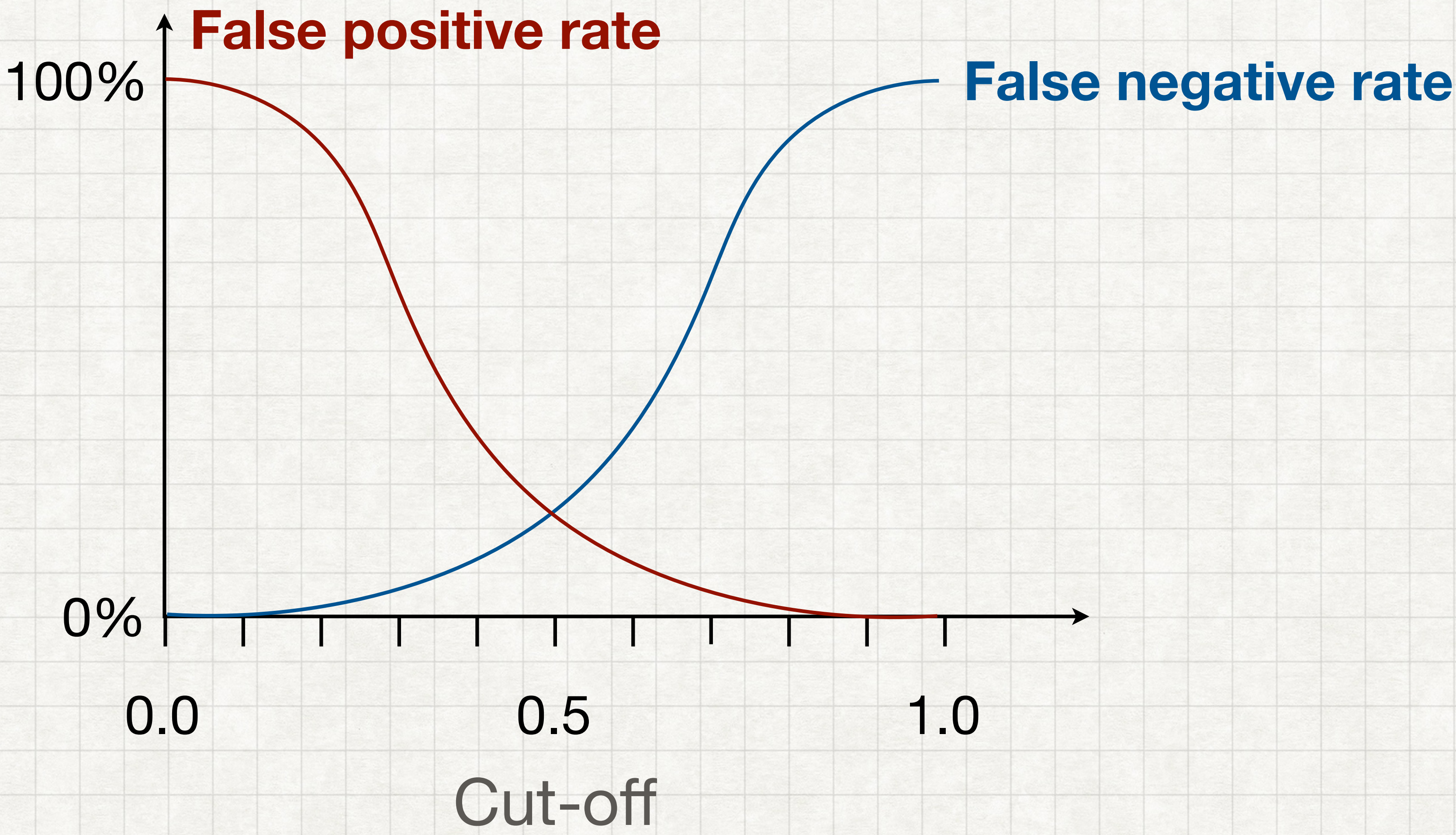
$$FPR = 990 / (990 + 8910) = 990 / 9900 = 10\%$$

$$FNR = 20 / (20 + 80) = 20 / 100 = 20\%$$

EXERCISE: Calculate proportion of items flagged by the detector that are actually defective

$$80 / (80 + 990) = 80 / 1070 \simeq 7.5\%$$

TRADE-OFF AS CUTOFF VARIES



Choosing cut-off depends on relative costs of true and false positives

PART II:
ONE-DIMENSIONAL ANOMALY DETECTION

EXAMPLE TRANSACTION STREAM

	id	category	price
0	710316821	QT	150.39
1	516025643	AA	346.69
2	414345845	QT	205.83
3	590179892	CB	55.61
4	117687080	QT	142.03
5	684803436	AA	152.10
6	611205703	QT	39.65
7	399848408	AA	455.67
8	289394404	AA	102.61
9	863476710	AA	297.82
10	534170200	KA	80.96
11	898969231	QT	81.39

CONSTRAINTS

- Field `id` (int): Identifier for item. Should not be null (`np.nan`), and should be unique in the table
- Field `category` (str): Should be one of "AA", "CB", "QT", "KA" or "TB"
- Field `price` (float): unit price in pounds sterling. Should be non-negative and no more than 1,000.00.

PYTHON2 USERS

Apparently, feather files are not compatible between Python2 and Python3:

```
if sys.version_info.major == 2:
    mv data data3
    mv data2 data
```

EXERCISE 1

EXPERT Find all the records that violate each constraint stated above.
You can get the data with:

```
from detect import get_2d_outlier_dataframe
df = get_2d_outlier_dataframe()
```

INTERMEDIATE Edit `exercise1.py` and try to uncomment the rows in main, one at a time, and fill in the parts marked `### FILL IN`.

There are hints, in order, in the file `hints.txt`.

Solution in `detect.py`

BEGINNER Run `detect.py`
Read through the code and try to understand how it works

SUPER EXPERT Find all bugs and other imperfections in `detect.py`

FINDING NULLS

```
def find_nulls(df, col):
    okcol = col + '_nonnull_ok'
    df[okcol] = df[col].notna()
    null_rows = df[df[okcol] == False]
    if len(null_rows) > 0:
        print(null_rows[['id', 'category', 'price']])
    else:
        print('No nulls in column %s' % col)
    return null_rows


def find_all_nulls(df):
    for c in list(df):
        find_nulls(df, c)
```

FINDING DUPLICATES

```
def find_dup_ids(df):
    df['id_unique_ok'] = (df.groupby('id')['id']
                            .transform('count') == 1)

    dup_ids = df[df.id_unique_ok == False]['id']
    if len(dup_ids) > 0:
        cid = dup_ids.groupby(df.id).count()
        cid.rename(columns={'id':'count'})
        cid.reset_index()
        print(cid)
    return dup_ids
```

FINDING OUTLIERS

```
def find_bad_prices(df):
    df['price_ok'] = np.logical_and(
                                df['price'] >= 0,
                                df['price'] <= 1000.0)
    bad_prices = df[df.price_ok == False]
    if len(bad_prices) > 0:
        print(bad_prices[['id', ‘category',
                                 'price']])
    return bad_prices
```

FINDING INVALID STRING VALUES

```
def find_bad_categories(df):
    allowed = ["AA", "CB", "QT", "KA", "TB"]
    df['category_ok'] = df['category'].isin(allowed)
    bad_cats = df[df.category_ok == False]
    if len(bad_cats) > 0:
        print(bad_cats[['id', 'category', 'price']])
    return bad_cats
```

FINDING ALL THE BADS

```
def find_bad_records(df):
    query = ('not (id_nonnull_ok '
                  'and id_unique_ok '
                  'and price_nonnull_ok '
                  'and price_ok '
                  'and category_nonnull_ok '
                  'and category_ok)')
    bads = df.query(query)
    print(bads)
    return bads
```

USING TDDA TO MAKE ALL THIS EASIER

TEST-DRIVEN DATA ANALYSIS

- Test-driven data analysis is methodology & software (open-source and commercial) for improving quality and robustness of analytical processes
- Two main components:
  - *Reference Testing:* extensions to **`unittest`** & **`pytest`** for testing analytical processes
  - *Automatic Constraint Discovery & Verification*.
- Can use the data verification capabilities as a general purpose anomaly detection framework, as we do here.
- We use the **`tdda`** library, available from PyPI.

EXAMPLE TRANSACTION STREAM

	id	category	price
0	710316821	QT	150.39
1	516025643	AA	346.69
2	414345845	QT	205.83
3	590179892	CB	55.61
4	117687080	QT	142.03
5	684803436	AA	152.10
6	611205703	QT	39.65
7	399848408	AA	455.67
8	289394404	AA	102.61
9	863476710	AA	297.82
10	534170200	KA	80.96
11	898969231	QT	81.39

CONSTRAINTS

- Field **`id`** (int): Identifier for item. Should not be null (`np.nan`), and should be unique in the table
- Field **`category`** (str): Should be one of "AA", "CB", "QT", "KA" or "TB"
- Field **`price`** (float): unit price in pounds sterling. Should be non-negative and no more than 1,000.00.

EXAMPLE TRANSACTION STREAM

```
{
  "fields": {
    "id": {
      "type": "int",
      "max_nulls": 0,
      "no_duplicates": true
    },
    "category": {
      "type": "string",
      "max_nulls": 0,
      "allowed_values":
        ["AA", "CB", "QT",
         "KA", "TB"]
    },
    "price": {
      "type": "real",
      "min": 0.0,
      "max": 1000.0,
      "max_nulls": 0
    }
  }
}
```

`constraints.tdda`

CONSTRAINTS

- Field **`id`** (int): Identifier for item. Should not be null (`np.nan`), and should be unique in the table
- Field **`category`** (str): Should be one of "AA", "CB", "QT", "KA" or "TB"
- Field **`price`** (str): unit price in pounds sterling. Should be non-negative and no more than 1,000.00.

ANOMALY DETECTION WITH TDDA

COMMAND-LINE VERSION

(Assuming numpy and pandas already installed!)

```
pip install tdda
pip install feather-format
pip install pmmif
cd pydatalondon2018ad/oned

tdda detect data/items.feather \    input data
            constraints.tdda   \    json constraints files
            bads.csv           \    location for output (.csv or .feather)
     --per-constraint          \    write cols for each constraint
     --output-fields                write all original cols too

cat bads.csv
```

CSV FILES IN TDDA LIBRARY

WE USE THESE OPTIONS FOR `pd.read_csv`

```
{
        'index_col': None,
        'infer_datetime_format': True,
        'quotechar': '"',
        'quoting': csv.QUOTE_MINIMAL,
        'escapechar': '\\',
        'na_values': ['', 'NaN', 'NULL'],
        'keep_default_na': False,
}
```

* *Pandas CSV reader does not readily allow empty strings and null values to co-exist when using blank for nulls.*

ANOMALY DETECTION WITH TDDA

API VERSION

```
from pmmif import featherpmm
from tdda.constraints import detect_df

path = 'data/items.feather'
df = featherpmm.read_dataframe(path).df

v = detect_df(df, ‘constraints.tdda',
              detect_per_constraint=True,
              detect_output_fields=[])
bads_df = v.detected()
print(bads_df)
```

GENERATING CONSTRAINT AUTOMATICALLY

COMMAND-LINE VERSION

```
tdda discover data/good_items.feather goods.tdda
```

or .csv
file containing known good (non-anomalous) data

location for output constraints file

or

```
tdda discover data/good_items.feather goods.tdda \
     --rex
```

Also generate regular expressions characterizing string fields

GENERATED CONSTRAINTS FILE

```
{
    "fields": {
        "id": {
            "type": "int",
            "min": 100000546,
            "max": 899995057,
            "sign": "positive",
            "max_nulls": 0,
            "no_duplicates": true
        },
        ⋮
        "price": {
            "type": "real",
            "min": 0.0,
            "max": 985.18,
            "sign": "non-negative",
            "max_nulls": 0
        }
    }
}
```

```
"category": {
        "type": "string",
        "min_length": 2,
        "max_length": 2,
        "max_nulls": 0,
        "allowed_values": [
            "AA",
            "CB",
            "KA",
            "QT",
            "TB"
        ],
        "rex": [
            "^[A-Z]{2}$"
        ]
    },
```

from `--rex`

GENERATING CONSTRAINTS AUTOMATICALLY

COMMAND-LINE VERSION

```
tdda discover data/good_items.feather goods.tdda
```

or .csv
file containing known
good (non-anomalous)
data

location for output
constraints file

or

```
tdda discover data/good_items.feather goods.tdda \
     --rex
```

Also generate regular expressions characterizing string fields

GENERATING CONSTRAINTS AUTOMATICALLY

API VERSION

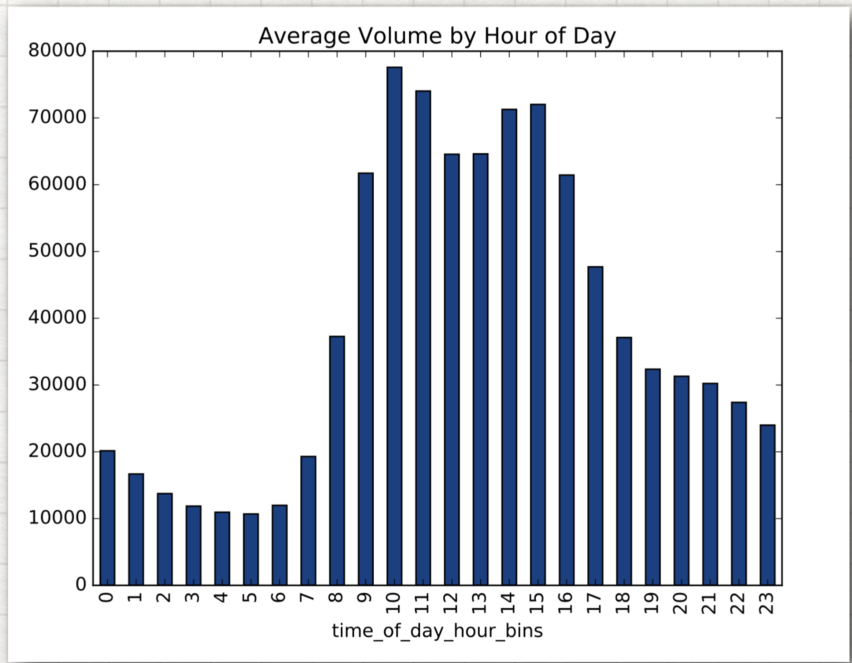
```
from pmmif import featherpmm
from tdda.constraints import discover_df

path = 'data/good_items.feather'
df = featherpmm.read_dataframe(path).df

constraints = discover_df(df, inc_rex=True)
with open('autoconstraints.tdda', 'w') as f:
    f.write(constraints.to_json())
```

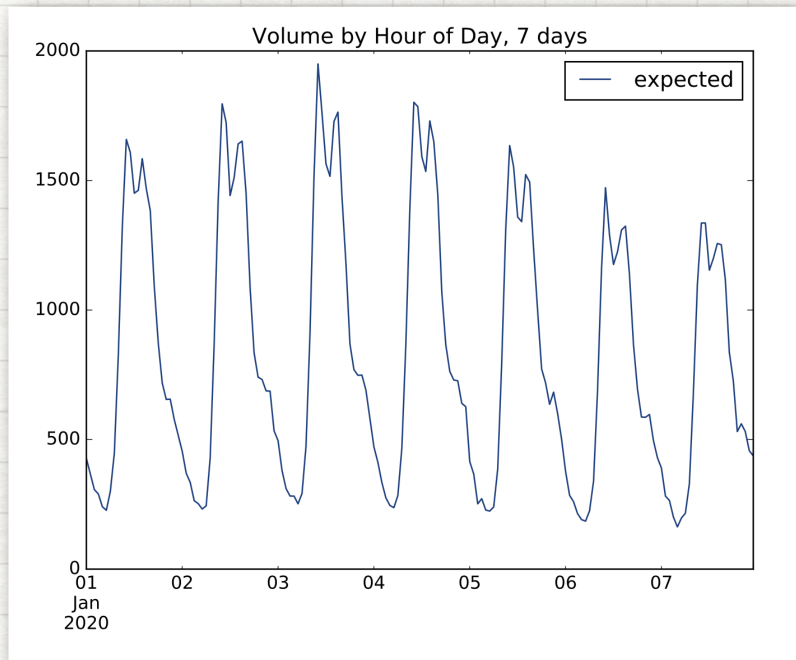
PART III: GENERALIZED SEASONALITY

GENERALIZED SEASONALITY

- With most data streams, there are regular, or semi-regular patterns in the data
    - month of year
    - day of week
    - time of day
    - "special" dates (public holidays)
    - important events (World Cup Matches, Parliamentary Debates, new releases of tdda library etc.)
- Relatedly, there are often long-term trends, such as growth rates
- In detecting anomalies, it's usually necessary to adjust for one or more of these.

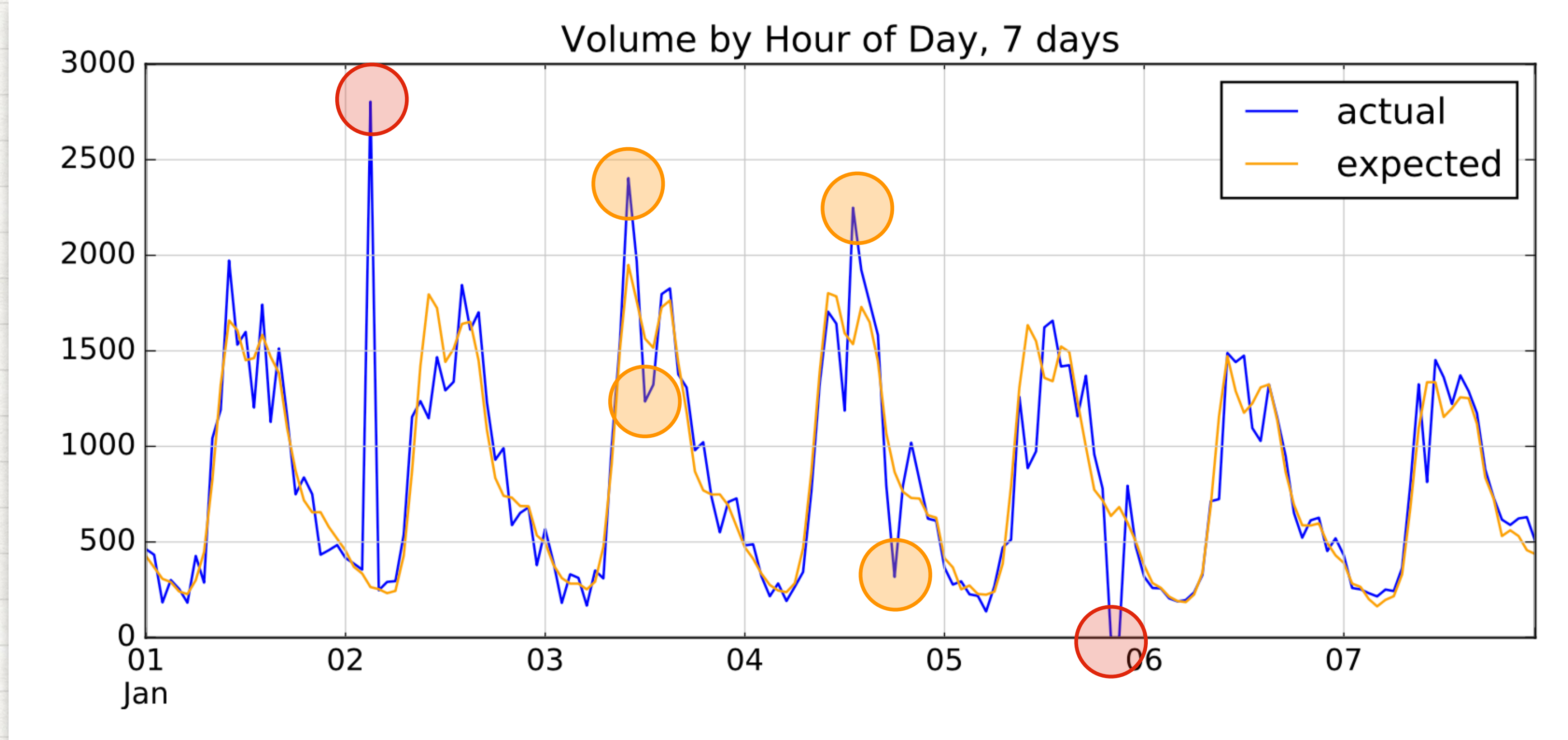
HOUR-OF-DAY, DAY-OF-WEEK





Typically collect each of these "background" patterns over many cycles to provide reference information for normalisation

ACTUAL VS. EXPECTED (NORMALIZED)



Two very obvious points that look anomalous here.

But what about the less clear-cut ones?

Usually need at least two criteria—proportionate and absolute deviation from expectation

EXAMPLE OUTLIER DEFINITION

Outlier if:

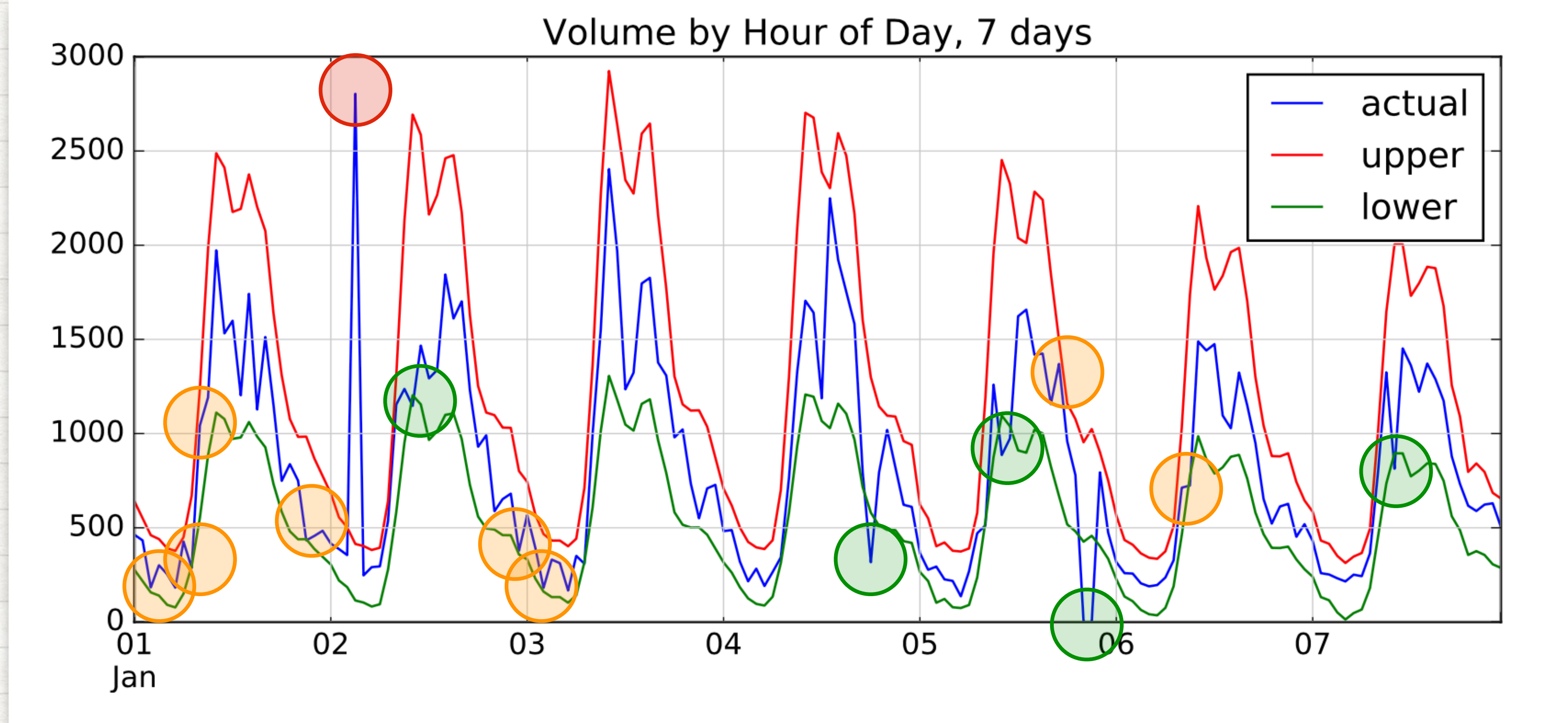
- actual value is larger than expected value by at least 50% and by at least 150;
- *or* actual value is lower than the expected by at least a third and by at least 150;

There are no general rules for choosing limits

Can use statistical measures (*n* standard deviations) etc., but the data usually being used for anomaly detection tends not to follow a pattern that makes this useful.

Normally just look at historical problems and set limits based on those, judgement and relative false positive/negative costs

ACTUAL VS. LIMITS



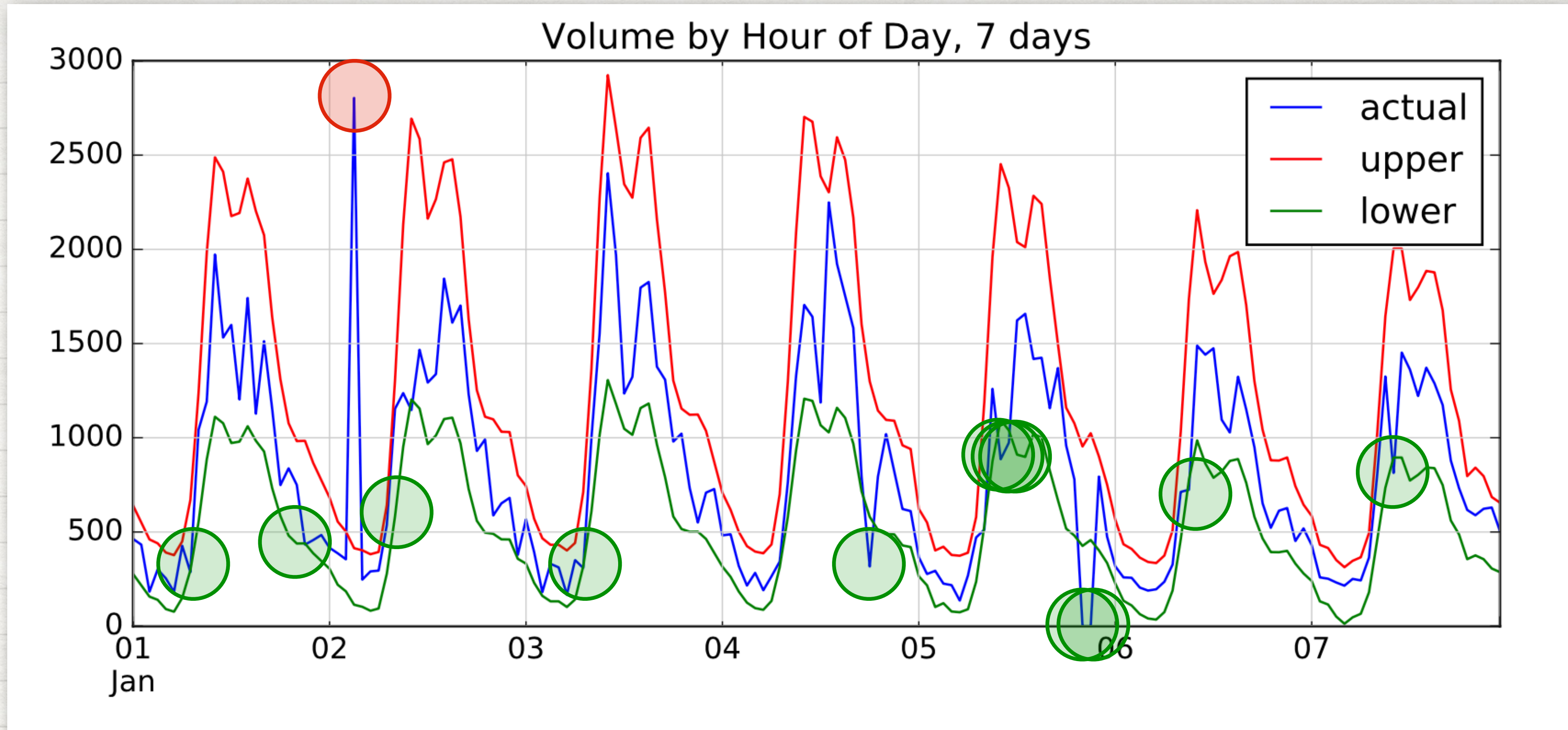

```
df['upper'] = np.maximum(df['expected'] * 1.5,
                         df['expected'] + 150)
df['lower'] = np.minimum(df['expected'] * 2 / 3,
                         np.maximum(df['expected']
                         - 150, 0))
```

ACTUAL VS. LIMITS

Can obviously use these directly to select anomalous records, or use them as columns for the tdda library to combine with others for overall anomaly detection:

```
df['actual_min_ok'] = df['actual'] >= df['lower']
df['actual_max_ok'] = df['actual'] <= df['upper']
```

date	actual	expected	upper	lower	actual_min_ok	actual_max_ok
2020-01-01 07:00:00	287	447	670.5	297.00	False	True
2020-01-01 21:00:00	433	656	984.0	439.52	False	True
2020-01-02 03:00:00	2804	264	414.0	114.00	True	False
2020-01-02 10:00:00	1147	1796	2694.0	1203.32	False	True
2020-01-03 07:00:00	309	475	712.5	318.25	False	True
2020-01-04 18:00:00	317	866	1299.0	580.22	False	True
2020-01-05 08:00:00	512	793	1189.5	531.31	False	True
2020-01-05 10:00:00	886	1635	2452.5	1095.45	False	True
2020-01-05 11:00:00	974	1551	2326.5	1039.17	False	True
2020-01-05 20:00:00	0	636	954.0	426.12	False	True
2020-01-05 21:00:00	0	683	1024.5	457.61	False	True
2020-01-06 09:00:00	724	1159	1738.5	776.53	False	True
2020-01-07 10:00:00	813	1336	2004.0	895.12	False	True



date	actual	expected	upper	lower	actual_min_ok	actual_max_ok
2020-01-01 07:00:00	287	447	670.5	297.00	False	True
2020-01-01 21:00:00	433	656	984.0	439.52	False	True
2020-01-02 03:00:00	2804	264	414.0	114.00	True	False
2020-01-02 10:00:00	1147	1796	2694.0	1203.32	False	True
2020-01-03 07:00:00	309	475	712.5	318.25	False	True
2020-01-04 18:00:00	317	866	1299.0	580.22	False	True
2020-01-05 08:00:00	512	793	1189.5	531.31	False	True
2020-01-05 10:00:00	886	1635	2452.5	1095.45	False	True
2020-01-05 11:00:00	974	1551	2326.5	1039.17	False	True
2020-01-05 20:00:00	0	636	954.0	426.12	False	True
2020-01-05 21:00:00	0	683	1024.5	457.61	False	True
2020-01-06 09:00:00	724	1159	1738.5	776.53	False	True
2020-01-07 10:00:00	813	1336	2004.0	895.12	False	True

EXERCISE 2

In `exercise2.py`, copy `plot_actual_vs_expected` as a new function `plot_actual_vs_limits`.

Calculate `upper` and `lower` cols, as on previous slides.

Uncomment the commented line to get a better colour scheme.

Delete or filter out the `expected` column from the DataFrame.

Uncomment the call in main and run it.

This should produce (the right) graph in `graphs/week-actual-vs-limits.svg`.

Hints in `hints2.txt`; solution is in `ad_norm_hour_day.py`.

EXERCISE 3

Identify the outliers algorithmically, rather than graphically and print them.

Hints in `hints3.txt`.

Solution in `ad_norm_hour_day.py`

njr@StochasticSolutions.com

http://stochasticsolutions.com

http://tdda.info

https://github.com/tdda

#tdda*

* *tweet (DM) us email address for invitation*
Or email me.

@tdda0 @njr0 @StochasticSolns

Correct interpretation: Zero (Error of interpretation: Letter "Oh")

https://github.com/tdda/pydatalondon2018ad/pydatalondon2018ad.pdf