Setting an upper limit on the branching ratio of FCNC $t\bar{t}$ processes with CMS data: a distributed approach

MAPD-B

FINAL PROJECT

DATA PROCESSING

14TH JULY 2021

Chiara Maccani
Samuele Piccinelli
Tommaso Stentella
Cristina Venturini

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

OVERVIEW

1. Our work case: FCNC $t\bar{t}$ processes
2. Our work environment and setting up Spark
3. Read and load data from S3 bucket
4. Performance study
5. Skimming and feature engineering
6. Making and plotting histograms
7. Statistical analysis
8. Conclusion and outlook

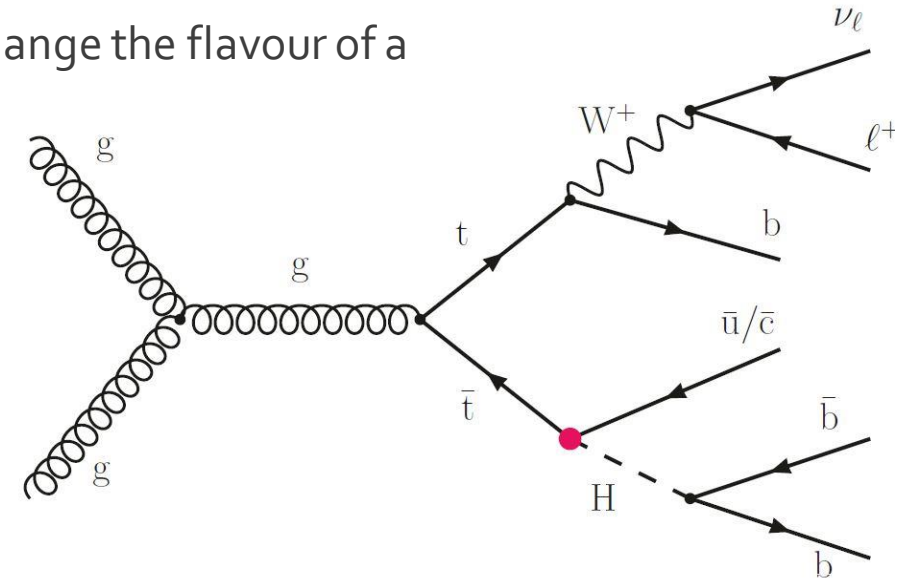
1 – FCNC $t\bar{t}$ PROCESSES

FCNC DECAY OF THE TOP QUARK TO THE HIGGS BOSON

Flavour changing neutral currents are weak interactions that change the flavour of a fermion without altering the electric charge.

Our analisys focuses on $t\bar{t}$ processes.

- **Standard Model**: $B(t \to Hq) = o(10^{-15})$ (absent at leading order and suppressed by GIM mechanism at higher orders)
- **Extension of SM** predict an enhanced $B(t \to Hq)$



➡ New physics?

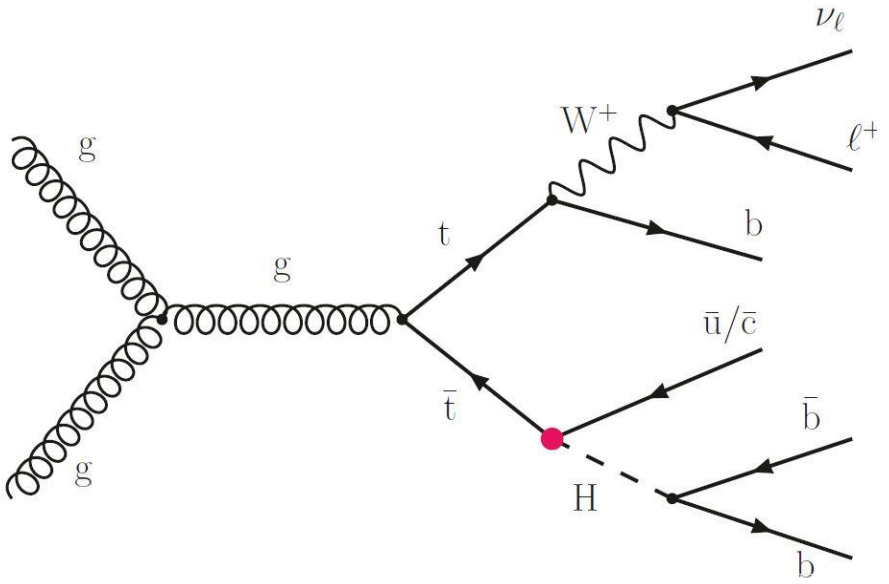
Main purpose: set an upper limit to the branching fraction of FCNC decay of the top quark in the trilepton channel $t\bar{t} \to Hq + Wb \to \mu\nu\ \mu\nu\ q + \mu\nu\ b$

FCNC DECAY OF THE TOP QUARK TO THE HIGGS BOSON

Trilepton channel: $t\bar{t} \rightarrow Hq + Wb \rightarrow \mu\nu\ \mu\nu\ q + \mu\nu\ b$

Final states:

- 3 muons (with isolation parameter < 0.15 and transverse momentum > 15 GeV)
- 0 isolated electrons electrons
- ≥ 2 jets
- ≥ 1 b-jets



Our data:

- Simulated FCNC samples
- Simulated Background: Single Top, Top-Top, Diboson, other processes (QCD, Drell Yan)
- CMS Data (Run 2016/2017/2018)

→ ~ 50 GB of file **.root** containing 19.000.000 events stored into a nested TTree structure

A distributed approach with **Spark** can help with the analysis

2 – OUR WORK ENVIRONMENT & SETTING UP SPARK

OUR WORK ENVIRONMENT & SETTING UP SPARK

```
Host gate
    HostName gate.cloudveneto.it
    Port 22
    User ******
    LocalForward 9998 10.67.22.59:22
    LocalForward 9997 10.67.22.232:22
    LocalForward 9996 10.67.22.233:22

Host master
    Hostname localhost
    User root
    Port 9998
    LocalForward 8900 localhost:8887
    LocalForward 8901 localhost:8080
    LocalForward 8902 localhost:4040
```

- 3 VM on CloudVeneto: one with 8 cores and 16 GB of memory, two with 4 cores and 6 GB of memory
- The first one employed as master, with two executors running
- The others employed as workers, with one executor each
- As a useful tool for an easy connection with the cluster from personal laptop a file `.ssh/config` has been used

3 – READ AND LOAD DATA FROM S3 BUCKET

READ AND LOAD DATA FROM S3 BUCKET

Data ingestion is the first stage of the process; files are stored in an Amazon S3 bucket.

An Amazon S3 bucket is a public cloud storage resource available in Amazon Web Services' (AWS) Simple Storage Service (S3). Here we read the ROOT file from S3 bucket into a Spark dataframe by reading the access ID and key from the `~/.s3cfg` file:

```
access_id = config.get('aws_profile', 'access_key')
access_key = config.get('aws_profile', 'secret_key')
```

READ AND LOAD DATA FROM S3 BUCKET

In order to use Hadoop, one needs to first configure it by creating a configuration object.

The configuration holds information about the job tracker, the input, output format and various other parameters.

It is necessary to properly configure and bootstrap Hadoop in order to submit the required jobs.

```
hadoop_conf=sc._jsc.hadoopConfiguration()

hadoop_conf.set('fs.s3n.impl', 'org.apache.hadoop.fs.s3native.NativeS3FileSystem')
hadoop_conf.set('fs.s3n.awsAccessKeyId', access_id)
hadoop_conf.set('fs.s3n.awsSecretAccessKey', access_key)
```

READ AND LOAD DATA

The data files are read into a Spark dataframe by means of a custom data loader (`readRoot()`).

Ad-hoc libraries are avoided:

- The DIANA-HEP library handles Spark DataFrame objects but is no longer mantained and poses compatibility problems between libraries;
- The ROOT.RDF.Experimental.Distributed Python library, which extends the functionalities of ROOT DF in a distributed environment;
- PyRDF, a pythonic wrapper around ROOT's RDataFrame with support for distributed execution.

All this solutions posed some techical issues to the implementation or were unsuited for the specific task.

READ AND LOAD DATA

We read the `.root` files in Numpy through the Uproot library.

Uproot is only an I/O library, primarily intended to stream data into machine learning libraries in Python.

We convert them in native Python types such as `lists`, `int`, `float`.


```
pro = uproot.open(file_name + ':Events').arrays(var, library='np')
```

READ AND LOAD DATA

Reading is done on the whole ~42 GB dataset, trying to avoid iterating over each different directory.

➡ each kind of sample is labelled with a different string: data, signal and a different tag for each MC process.

Finally, we instantiate the RDD:

```
dfRoot = sc.parallelize(fileList).flatMap(readRoot).toDF()
```

4 – PERFORMANCE STUDY

4 - PERFORMANCE STUDY

We briefly investigated how performance changed when varying:

- Number of executors
- Cores per executor
- Memory per executor
- Partitions of the files

There's no golden rule, not even a good rule of thumb, to fix these parameters appropriately.

Most of the decisions are made based on the exact kind of hardware one is working with and by simply investigating which configuration gives the best performance.

⟹ One heuristic rule, found in literature, suggests to have a number of partitions equal to 3-4 times the available cores for each executor. Also, it is suggested to adopt a partition size of around 130 MB.

4 - PERFORMANCE STUDY

Accordingly with these considerations, we test the following cores-executors-memory configurations

Code	Configuration	Memory
a	16 exe, each w/ 1 core	1.5 Gb
b	8 exe, each w/ 2 cores	3 Gb
c	3 exe, 1 w/ 4 cores, 2 w/ 2 cores	6 Gb
d	4 exe, each w/ 4 cores	6 Gb
e	3 exe, 1 w/ 8 cores, 2 w/ 4 cores	6 Gb

For each of them, we vary the number of partitions in the set [4, 12, 24, 36, 48, 56, 64] and test their performance by measuring the execution time of a count() action on one of the background directories 'tZq' (size: 4.5 GB).

4 - PERFORMANCE STUDY

To perform our evaluation of the elapsed time, we made use of a library called **SparkMeasure,** which is a tool for performance troubleshooting of Apache Spark jobs.

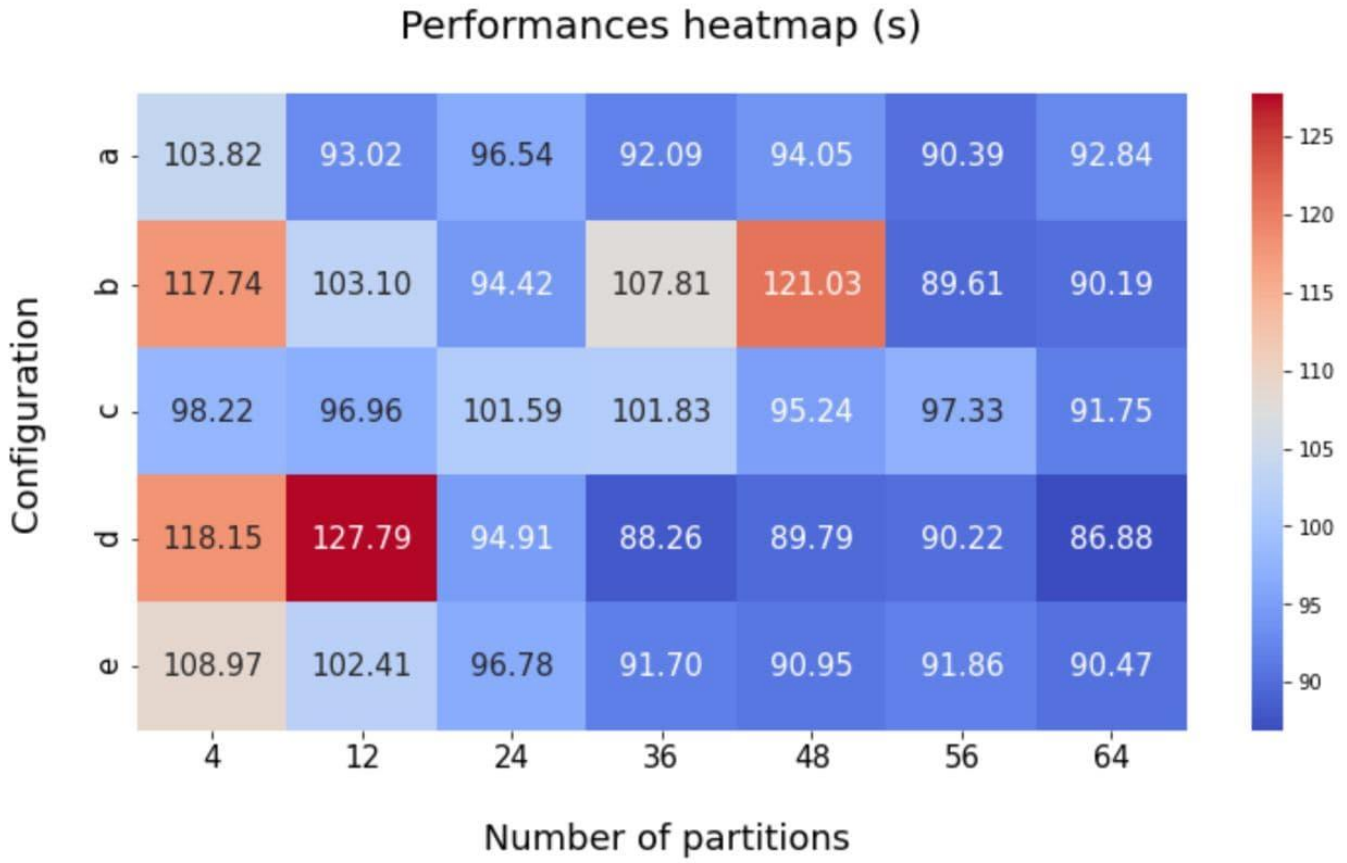
It simplifies the collection and analysis of Spark performance metrics.

The following function prints the metrics of the Spark job to a json file.

```
from sparkmeasure import StageMetrics
stagemetrics = StageMetrics(spark)

def sparkmeasure(df, idx, ex):
        stagemetrics.begin()
        df.count()
        stagemetrics.end()

        df_out = stagemetrics.create_stagemetrics_DF('PerfStageMetrics’)
        stagemetrics.save_data(df_out, './perf_test/stagemetrics_{}_{}'.format(ex, idx))
```

We then retrieve the results and access the elapsed time, the entry we analyzed.

4 - PERFORMANCE STUDY



As one can see, apart from some configurations which perform evidently worse than the others (20-30 seconds more), almost all the others employ a similar execution time.

4 - PERFORMANCE STUDY

The best configuration is *d* with 36 number of partitions, which agrees with the heuristic rule found in literature.

Nevertheless, by almost doubling the number of partitions, performance remains practically identical.

5 – SKIMMING AND HIGH LEVEL FEATURES

5.1 - SKIMMING

After loading the data, the next step consists in skimming the dataframes:

- We apply filters to retain the events we are interested in
- We create new columns to aide us in the subsequent step of creating high level features

To do so, we define a function skimmer(df) which uses 3 Spark transformations:

- `withColumn()`, to define new columns containing:
  - masks based on specific conditions
  - new variables
- `filter()`, to discard events not respecting certain conditions
- `drop()`, to drop useless columns

5.1 - SKIMMING

In particular, inside `withColumn()`, we call a series of UDFs (User Defined Function). The decorator is imported from the correspondent PySpark library.

```
import pyspark.sql.functions as pf
import pyspark.sql.types as pt

arr = pt.ArrayType
maskMu = pf.udf(lambda iso, pt: [True if i < .15 and p > 15 else False for i,p in (iso,pt)],
    arr(pt.BooleanType()))
maskEl = pf.udf(lambda iso: [True if i < .15 else False for i in iso], arr(pt.BooleanType()))
apply_mask_long = pf.udf(lambda x, mask: [i for i,m in (x,mask) if m], arr(pt.LongType()))
apply_mask_doub = pf.udf(lambda x, mask: [i for i,m in (x,mask) if m], arr(pt.DoubleType()))
count_col = pf.udf(lambda m: ((m)), pt.IntegerType())
```

When working with UDFs it's important to specify the correct return type, otherwise an error will arise. To do so, we use the appropriate library.

5.2 – HIGH LEVEL FEATURES

The next step in the analysis consists in using the output of the previous function to construct some higher level features, specifically:

- $\Delta\varphi$
- $\Delta R = \sqrt{(\Delta\eta)^2 + (\Delta\varphi)^2}$
- Invariant mass between each pairing of leptons
- Invariant mass between all lepton in an event (3)

These are the variables we hope to be discriminant, in order to be able to use them to set the limit on the branching ratio.

They also serve to further filter our data, imposing conditions which help to discard additional useless events.

5.2 – HIGH LEVEL FEATURES

All the functions we use to define these new variables are UDFs , built by us:

- `invMass(pt, eta, phi, mass)`, for the invariant mass of both 2 and 3 particles.
- `DeltaR(eta1, eta2, phi1, phi2)`, to compute ΔR
- `find_idx(charge, phi, eta)`, to order the leptons in the correct way (first the lepton of opposite sign, then the other two in increasing order of angular separation ΔR

The output DataFrame contains new columns with these new variables, which are all flat.

At this point, we are still retaining some non-flat columns, simply because we wish to plot them.

Nevertheless, one could easily drop all the non-flat columns and proceed with a flattened DataFrame.

6. MAKING AND PLOTTING HISTOGRAMS

HISTOGRAMS

Goal : plot the stacked backgrounds' histograms , overlap data, plot the shape of simulated signal

➡ Parallelize the creation of histograms, by mapping functions on the rows and splitting into 3 flows (data, signal, bkg) based on 'sample'

A **Map – Reduce By Key** approach is used

```
histos = dict(
    data.select([variable, 'eventWeightLumi', 'sample'])
        .rdd
        .map(lambda  x: histo(x[ variable], x['eventWeightLumi'], x['sample'],

                    range_ = interval, bins = nbins) )
        .reduceByKey(lambda x,y : (x[0], x[1]+y[1]))
        .take(50)
    )
```

```
import libraries and useful dependencies
```

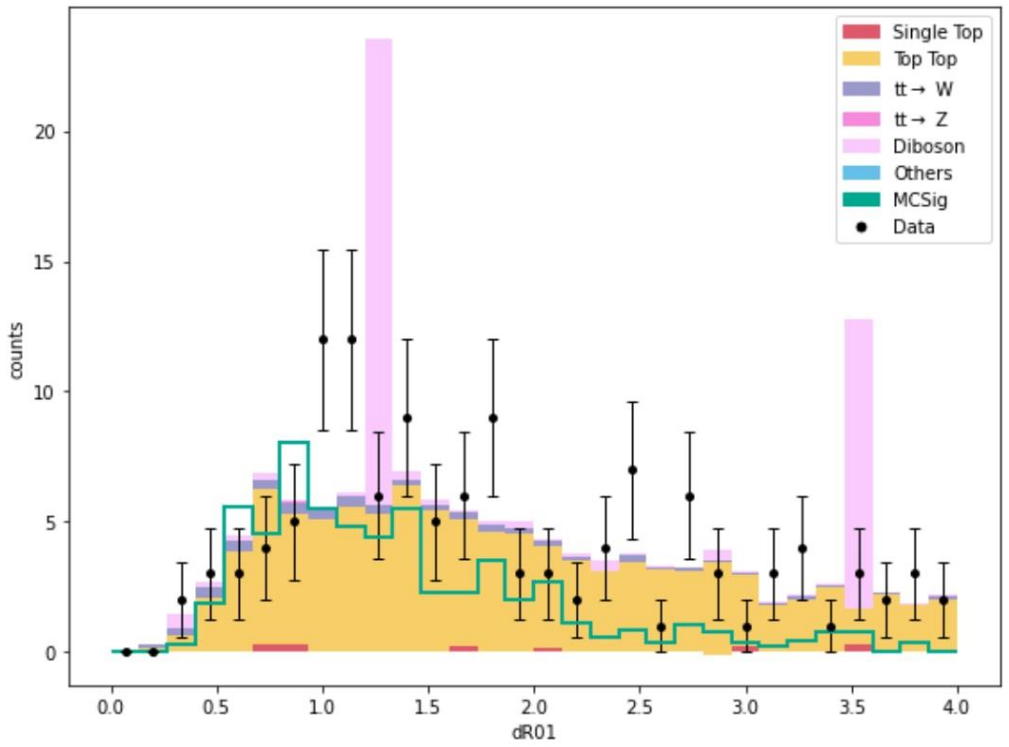
HISTOGRAMS

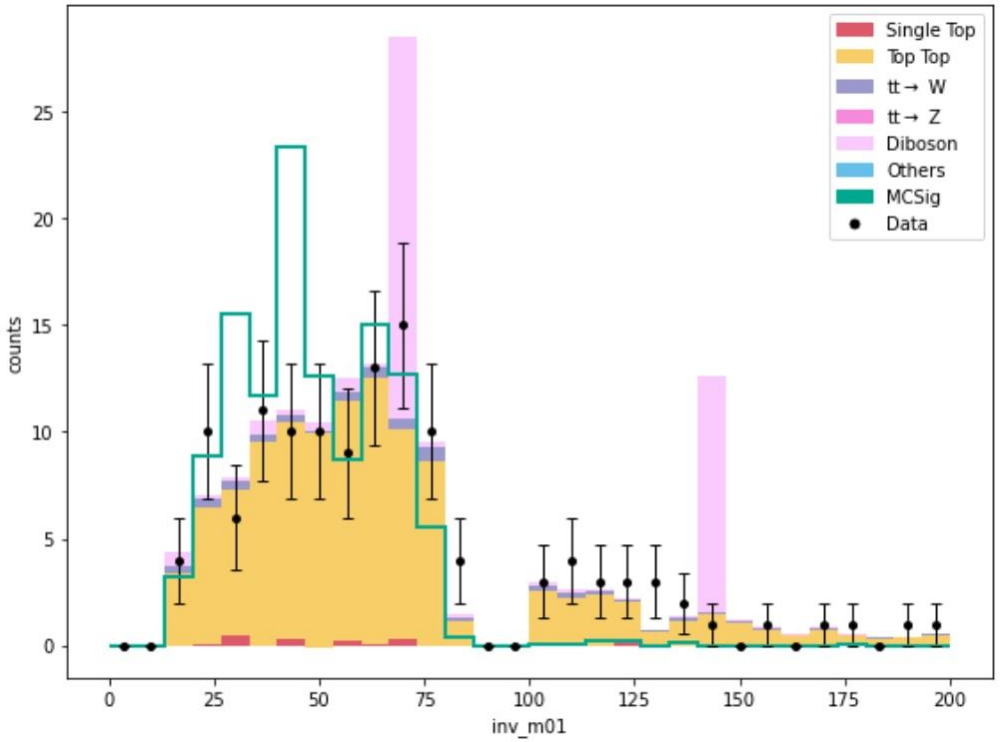
RDD object has a function **.histogram()** but it does not take into account weights

➡ We decide to implement our own function

```
def histo (variable, weight, sample, range_, bins):
    …
    computes the edges of bins, scans the limit's couples and looks for the bin in which
    falls value of the variable in the row in
   examination, then multiplies for the weight.
   counts  is a list of length bins of the type [0, 0, … , weight, 0, 0]
    ...
    return (sample, [edges, counts])
```

HISTOGRAMS


Single Top
Top Top
tt→ W
tt→ Z
Diboson
Others
MCSig
Data
counts
dR01
0
5
10
15
20
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0


Single Top
Top Top
tt→ W
tt→ Z
Diboson
Others
MCSig
Data
counts
inv_m01
0
5
10
15
20
25
0
25
50
75
100
125
150
175
200

7 – STATISTICAL ANALYSIS

STATISTICAL ANALYSIS

- The entire procedure is based on the informations about the histograms computed in the previous step, that is the number of bins and the counts in each of them: no need for caching large amount of data.
- The parallelization begins with the generation of toys, proceeds with the computation of the likelihood and ends with the computation of the q_μ distributions
- For every bin `ntoys` data must be extracted with the constraint of having a certain set of `ntoys` data binded to a given bin in all the computed likelihood: this contrains the structure of the RDD of the seeds.
- A like function is defined to be used in a map trasformation to compute the likelihood

```
def like(x, mu_sig, bkg, mu_best):
    from prod.stats import pisson as ps
    return np.prod(ps.pmf(x, mu_sig)), np.prod(ps.pmf(x, bkg)), np.prod(ps.pmf(x, mu_best))
```

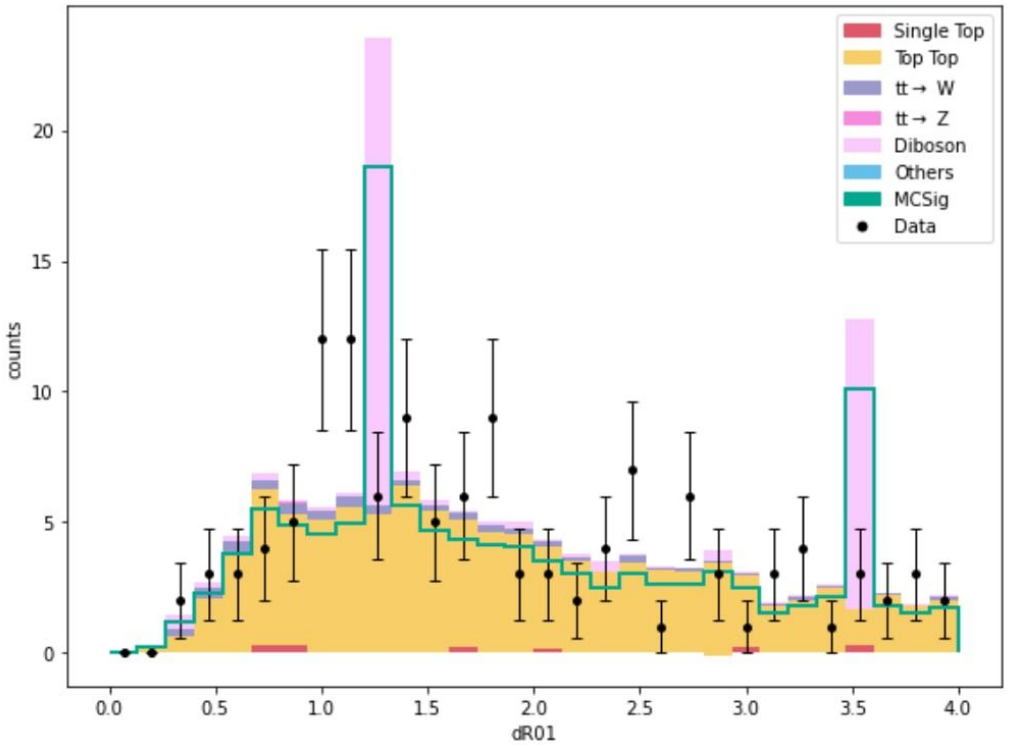
STATISTICAL ANALYSIS

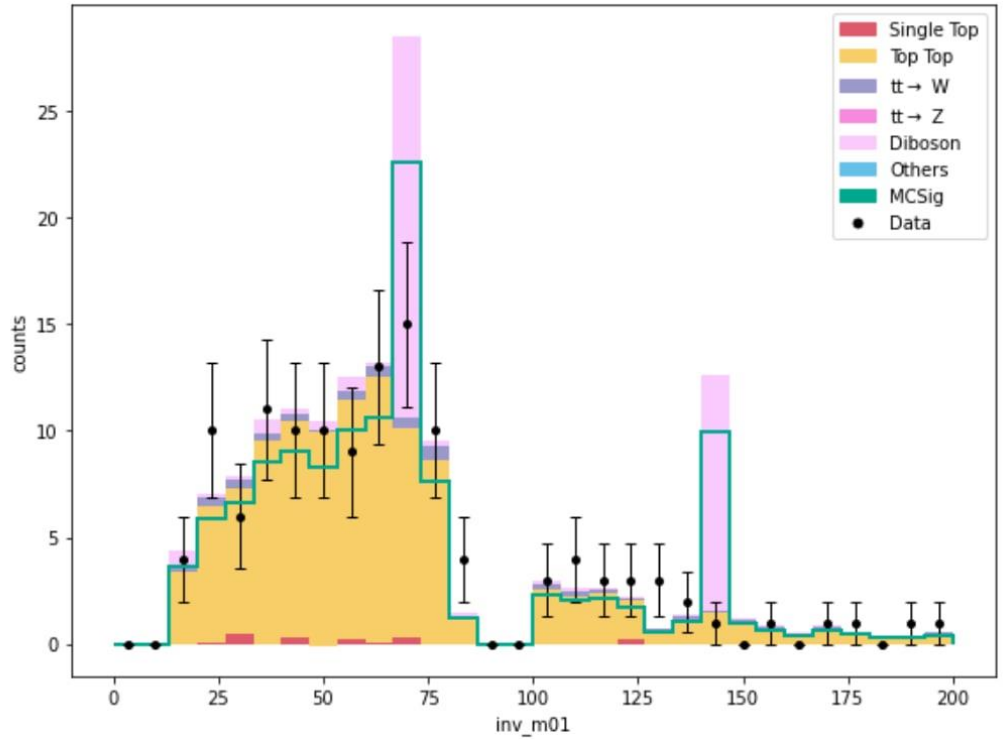
- Two aim is to parallelize the generation of the toys data
- Two possible approaches:
  - Extreme parallelization: all the ntoys × nbin in a single list parallelized to RDD, then reduce by key to reobtain the binding with bins.
    - Not effective parallelization: too many seeds with respect to the number of cores, does not exploit optimized vectorization with a single core.
  - The best approach is to split the number of toys to be generated in smaller chunks which can be effectively manipulated in a vectorized way by a single core thanks to libraries like Scipy.

STATISTICAL ANALYSIS

```
toys = sc.parallelize(
            np.concatenate((np.tile(np.arange(0,ntoys,1),(nbin,1)).T.reshape(ntoys*nbin,1),
                np.tile(b_plus_sig_mu,(ntoys,1)).reshape(ntoys*nbin,1)),axis=1))\
            .map(lambda x: (x[0], ps.rvs(x[1]))).reduceByKey(lambda x,y: flatten(x,y))\
            .map(lambda x: np.array(x[1])
            )

toys = sc.parallelize(
            np.tile(b_plus_sig_mu, (ntoys,1))).map(lambda bda x: ps.rvs(x)
            )

like_tot = toys.map(lambda x: like(x, b_plus_sig_mu, bkg, b_plus_sig_best))

q_mu = np.array(like_tot.map(lambda x: (-2*np.log(x[1]/x[2]), -2*np.log(x[0]/x[2]))).take(ntoys))
```

FINAL HISTOGRAMS





8 – CONCLUSIONS AND OUTLOOK

CONCLUSION AND OUTLOOK

A search for flavor-changing neutral currents events via distributed computing is here presented.

The main advantage lies in a faster access, filtering and parallelized HLF creation: computations are optimized, yet a noticeable advantage wrt. the standardized approach is not significant given the reduced size of the cluster.

Main difficulties:

- Non-tabular nature of the features stored in `.root` files;
- No standard data loader available;
- I/O operations bounded by the network bandwidth in CV.

Future prospects:

- Extend the analysis to the other iSkim categories;
- Run the code on a larger cluster.

Thanks for the attention!