JMCTF Documentation

Release 1.0.0

Ben Farmer

Oct 21, 2020

Contents:

Python tools for perfoming Monte-Carlo studies of joint distribution functions consisting of many independent components, powered by TensorFlow.

CHAPTER 1

Installation

The recommended way to install the package is from PyPI, i.e.:

```
pip install jmctf
```

However this project is hosted on github, so you can also clone the repository and install from that in the usual ways, e.g.:

```
git clone https://github.com/bjfar/jmctf.git
pip install ./jmctf
```

Even when installing from the git repo, it is still recommended to use pip since this is compatible with anaconda environments.

Once installed the package can be imported in Python using the module name `jmctf`.

CHAPTER 2

Quick Start

The principal pipeline which JMCtf is designed to streamline is the following:

1. *Combine independent "analyses" into a joint distribution*
2. *Sample from the joint distribution*
3. find_MLEs_long
4. build_test_stats_long

Getting more advanced, we can loop over the whole procedure to simultaneously test many alternate hypotheses and compute trial_corrections, as well as define new analysis classes to extend JMCtf with new component probability distributions.

A fast introduction to the package, then, is to see an example of this pipeline in action. So let's speed through this! For details please see the longer descriptions given at the links above (or under the *Basic usage* header in the navigation bar).

First, we create some "Analysis" objects and combine them into a `JointDistribution` object for joint treatment:

```
>>> import numpy as np
>>> import tensorflow as tf
>>> from JMCTF import NormalAnalysis, BinnedAnalysis, JointDistribution
>>> norm = NormalAnalysis("Test normal", 5, 2) # (name, obs., std.dev.)
>>> bins = [("SR1", 10, 9, 2), # (name, obs., b, sigma_b)
...         ("SR2", 50, 55, 4)]
>>> binned = BinnedAnalysis("Test binned", bins)
>>> joint = JointDistribution([norm,binned])
```

Next, set some input (e.g. "null") parameters for the analyses:

```
>>> free, fixed, nuis = joint.get_parameter_structure()
>>> print("free:", free)
{'Test normal': {'mu': 1}, 'Test binned': {'s': 2}}
```

(continues on next page)

(continued from previous page)

```
>>> null = {'Test normal': {'mu': [[0.]], 'nuisance': None}, 'Test binned': {'s': [[0.
↪, 0.]], 'nuisance': None}}
>>> joint_null = joint.fix_parameters(null)
```

and sample from the joint distribution:

```
>>> samples = joint_null.sample(1e6)
>>> q_null, joint_fitted_null, all_pars_null = joint_null.fit_all(samples)
# Inspect shapes
>>> print({k: v.shape for k,v in samples.items()})

{'Test normal::x_theta': TensorShape([1000000, 1]),
 'Test normal::x': TensorShape([1000000, 1]),
 'Test binned::x': TensorShape([1000000, 1, 2]),
 'Test binned::n': TensorShape([1000000, 1, 2])}

# Plot all sample distributions
>>> import matplotlib.pyplot as plt
>>> from JMCTF.plotting import plot_sample_dist
>>> fig = plot_sample_dist(samples)
>>> plt.show()
```

Now let us find the maximum likelihood estimators for all parameters in the *JointDistribution* for each of these samples, and plot those too:

```
>>> from JMCTF.plotting import plot_MLE_dist
>>> q_fit, joint_fitted_to_null_samples, all_pars_fit = joint_null.fit_all(samples)
>>> fig = plot_MLE_dist(all_pars_fit)
>>> plt.show()
```

And the distribution of a log-likelihood-ratio test statistic:

```
>>> q_null = -2*joint_null.log_pdf(samples)
>>> LLR = q_null - q_fit
>>>
```

Combine independent "analyses" into a joint distribution

moved

Sample from the joint distribution

moved

2.1 Fit MLEs to samples under a hypothesis (or many hypotheses)

In the above example, we generated samples from a distribution that was fit to some data that we manually invented. But that is a weird thing to do. More conventionally we want to generate samples under a pre-defined *null hypothesis*, and use them to understand the distribution of some test statistic under that hypothesis. To do this, it is the parameter values that we want to fix manually, not the samples. We can supply these to the constructor of a *JointDistribution* object, however we can also introspect a parameter-less *JointDistribution* to determine what parameters are required in the event that we want to fix them:

```
>>> free, fixed, nuis = joint.get_parameter_structure()
>>> print(free)
{'Test normal': {'mu': 1}, 'Test binned': {'s': 2}}
```

As with the sample structure introspection, this dictionary tells us which free parameters we need to supply, and what their dimension should be. The other dictionaries, *fixed* and *nuis*, tell us the structure of *fixed* and *nuisance* parameters respectively. The analyses themselves know sensible null hypothesis values for these, so we don't have to supply them, though we do have to explictly ask for the default values by setting the 'nuisance' dictionary key to *None*. See **TODO** for more details.

So, let us define a null hypothesis, sample from it, and then find MLEs for all parameters for all those sample:

```
>>> #TODO: Internally ensure float32 format to avoid this verbosity?
>>> null = {'Test normal': {'mu': np.array([[0.]],dtype='float32'), 'nuisance': None},
...         'Test binned': {'s': np.array([[0., 0.]],dtype='float32'), 'nuisance':␣
↪None}}
>>> joint_null = joint.fix_parameters(null)
>>> # or alternatively one can supply parameters to the constructor:
>>> # joint_null = JointDistribution([norm,binned],null)
>>> samples = joint_null.sample(3)
>>> q_null, joint_fitted_null, all_pars_null = joint_null.fit_all(samples)
>>> print(to_numpy(all_pars_null))
{'Test normal':
 {'mu': array([[[-2.4318216]],
              [[ 0.7213395]],
              [[-1.833349 ]]], dtype=float32),
  'theta': array([[[0.]],
                  [[0.]],
                  [[0.]]], dtype=float32),
  'sigma_t': 0.0},
 'Test binned':
 {'s': array([[[ 0.68332815,  1.2073289 ]],
              [[-0.5019169 ,  0.8478795 ]],
              [[ 1.0781676 , -0.9483687 ]]], dtype=float32),
  'theta': array([[[-0.7318874, -1.5432839]],
                  [[-0.0951565, -1.0360898]],
                  [[-1.4436939,  1.2477738]]], dtype=float32)}}
>>> # Inspect shapes
>>> print({k1: {k2: v2.shape for k2,v2 in v1.items()}
...  for k1,v1 in to_numpy(all_pars_null).items()})
{'Test normal': {'mu': (3, 1, 1),
                 'theta': (3, 1, 1),
                 'sigma_t': ()},
 'Test binned': {'s': (3, 1, 2),
                 'theta': (3, 1, 2)}}
```

We now need to start discussing the input/output array shapes more carefully. You will notice that we have supplied the parameters as two-dimensional arrays, even though it seems like one dimension should be enough. This is because we can use the *JointDistribution* class to collect and fit samples from *many* hypothesis simultaneously. That is, we could supply many alternate input parameters at once (we will do this in section **TODO**). On the output side, the structure of the fitted MLEs also reflects this possibility. Take for example the shape of the MLE for the parameter *'s'* from the *Test binned* analysis. This is *(3, 1, 2)*: 3 is the sample dimension (we have three samples per hypothesis to fit), 1 is the hypothesis dimension (we only provided one input hypothesis), and 2 is the fundamental dimension of *'s'* (we have two bins in the analysis, each characterised by a single parameter).

For simplicity, JMCTF is restricted to this three-dimensional form. Only one dimension for hypotheses, one dimension for samples, and one dimension for parameters is permitted. If you want to create e.g. a matrix of parameters in a custom Analysis class, it will need to be flattened when returning it to higher JMCTF classes, and if you want a multidimensional array of samples then you will need to sample them in 1D and then reshape. Likewise, arrays of input hypotheses will need to be appropriately flattened.

TODO refer to more detailed shape discussion elsewhere?

2.2 Build and analyse test statistics

Now that we understand the basic machinery, we can start to do some statistics!

CHAPTER 3

Basic usage

In these sections you can find explanations and demonstrations on the basic usage of JMCTF to perform Monte Carlo analysis of likelihood-based test statistics. The structure loosely follows that of the *Quick Start* guide, but with more details.

3.1 Combine independent "analyses" into a joint distribution

The basic components in JMCTF are "analyses". These are objects describing the joint PDF of a single "analysis" or "experiment", for example an experiment searching for a signal in a series of Poisson "bins" or signal regions, or simply a normal random variable representing an experimental measurement. But these "analysis" objects do more than just describe the PDF of the experiment; they are responsible for constructing the TensorFlow probability model for the experiment (basically the PDF), for providing good starting guesses for distribution parameters based on sample realizations, for scaling parameters such that their MLEs have approximately unit variance (to help out the TensorFlow minimisation routines), for providing Asimov samples, and for keeping track of the sample structure. *Extending JMCTF with new analysis objects* is somewhat involved so it is covered separately.

JMCTF is designed for jointly analysing many independent experiments simultaneously. So to begin, let us create two simple experiments. First, a simple normal random variable, characterised by a mean and standard deviation. This can be created from the `NormalAnalysis` class:

```
>>> from JMCTF import NormalAnalysis, BinnedAnalysis, JointDistribution
>>> norm = NormalAnalysis("Test normal", 5, 2)
```

where the arguments are simply a name for the experiment, followed by an "observed" value, and the standard deviation of the normal random variable. The mean is treated as a free parameter so it is not specified. The "observed" value will only be used for final p-value computations, after all simulations are done, so it does not affect the simulations themselves.

Next, an experiment consisting of several independent Poisson random variables, whose means are characterised in terms of a "signal" parameter (either fixed, or to be fitted) and a second "background" parameter constrained by a normally distributed control measurement:

```
>>> bins = [("SR1", 10, 9, 2),
...         ("SR2", 50, 55, 4)]
>>> binned = BinnedAnalysis("Test binned", bins)
```

where the bin data consists of a name, an "observed" value, an expected background parameter, and an uncertainty parameter for the background. A full description is given in the `BinnedAnalysis` class documentation.

Not much can be done with these Analysis objects on their own; they act mainly as "blueprints" for experiments, to be used by other classes. The simplest such class is `JointDistribution`, which can do such things as jointly fit the parameters of many Analysis classes, and sample from the underlying distribution functions[1].

Creating a joint PDF object with `JointDistribution` is as simple as providing a list of component Analysis objects to the constructor:

```
>>> joint = JointDistribution([norm,binned])
```

Fitting the joint PDF to some data requires that the data be provided in an appropriately structured dictionary, so that samples can be assigned to the correct Analysis. The correct structure to use can be revealed by the `get_sample_structure` method:

```
>>> joint.get_sample_structure()
{'Test normal::x': 1, 'Test normal::x_theta': 1, 'Test binned::n': 2, 'Test binned::x
↪': 2}
```

Here the dictionary keys are in the format *analysis_name::random_variable_name* and the values give the dimension of those random variables. For example we defined two bins in our `BinnedAnalysis`, so the dimension of the bin count random variable *Test binned::n*, and control variable *Test binned::x*, is 2.

Knowing this information, we can manually construct a sample for the full joint PDF and fit the joint PDF to it:

```
>>> my_sample = {'Test normal::x': 4.3, \
...              'Test normal::x_theta': 0, \
...              'Test binned::n': [9,53], \
...              'Test binned::x': [0,0]}

>>> q, joint_fitted, par_dicts = joint.fit_all(my_sample)
>>> print(par_dicts["all"])
{'Test normal':
 {'mu': <tf.Variable 'mu:0' shape=() dtype=float32, numpy=4.3>,
  'theta': <tf.Variable 'theta:0' shape=() dtype=float32, numpy=0.0>,
  'sigma_t': <tf.Tensor: id=57, shape=(), dtype=float32, numpy=0.0>},
'Test binned':
 {'s': <tf.Variable 's:0' shape=(2,) dtype=float32, numpy=array([ 0.        , -0.
↪23735635], dtype=float32)>,
  'theta': <tf.Variable 'theta:0' shape=(2,) dtype=float32, numpy=array([0., 0.],␣
↪dtype=float32)>}}
```

The output is not so pretty because the parameters are TensorFlow objects. We can convert them to numpy with the `common.to_numpy()` function for better viewing:

```
>>> from JMCTF.common import to_numpy
>>> print(to_numpy(par_dicts["all"]))
{'Test normal':
  {'mu': 4.3, 'theta': 0.0, 'sigma_t': 0.0},
 'Test binned':
```

(continues on next page)

[1] As you might guess from the name, the `JointDistribution` class inherits from `tensorflow_probability.JointDistributionNamed`. So all the standard log probability and sampling methods etc. from tensorflow_probability are available.

(continued from previous page)

```
  {'s': array([ 0.        , -0.23735635], dtype=float32),
   'theta': array([0., 0.], dtype=float32)}}
```

The `fit_all` method fits all the free parameters in the full joint distribution to the supplied samples, and returns *q* (negative 2 times the log probability density for the samples under the fitted model(s)), a new *JointDistribution* object fitted to the samples (the original remains unchanged), and three dictionaries of parameter values ("all" parameters, just the "fitted" parameters", and just the "fixed" parameters) packed into a dictionary.

Here we have only fit the PDF to one sample, however the power of JMCTF really lies in fitting the PDF to lots of samples quickly. Of course manually creating lots of samples is tedious and not very useful; a more standard workflow is to actually *sample* the samples from the PDF under some fixed "null hypothesis" parameter values. We cover this in *Sample from the joint distribution*.

(Note, the crappy formatting of these footnotes is fixed in later versions of the sphinx_rtd_theme, and should go away as soon as readthedocs adopt a new release of it)

3.2 Sample from the joint distribution

In the previous section (*Combine independent "analyses" into a joint distribution*), having fitted our joint PDF to a single sample, we obtained a second *JointDistribution* object as output, whose parameters are set to their maximum likelihood estimators given that sample. Its probability distribution is therefore now fully-specified, and we can sample from it:

```
>>> samples = joint_fitted.sample(3)
>>> print(to_numpy(samples))
{'Test normal::x_theta': array([0., 0., 0.], dtype=float32),
 'Test normal::x': array([10.776527 , 10.519511 ,  9.5911875], dtype=float32),
 'Test binned::x': array([[[-3.0337536, -1.6310872]],
                          [[-4.0343146,  1.5532861]],
                          [[-2.800459 , -0.7948484]]], dtype=float32),
 'Test binned::n': array([[[ 8., 40.]],
                          [[ 2., 50.]],
                          [[ 7., 60.]]], dtype=float32)}
```

As before, we can fit our distribution to these samples, and this time we will obtain maximum likelihood estimators for each sample independently:

```
>>> q_3, joint_fitted_3, par_dict_3 = joint.fit_all(samples)
>>> print(to_numpy(par_dict_3["all"]))
{'Test normal':
 {'mu': array([10.776527 , 10.519511 ,  9.5911875], dtype=float32),
  'theta': array([0., 0., 0.], dtype=float32),
  'sigma_t': 0.0},
 'Test binned':
 {'s': array([[[ 0.5640617 , -1.586598  ]],
              [[-0.82253313, -0.7777322 ]],
              [[ 0.22200736,  0.68772215]]], dtype=float32),
  'theta': array([[[-1.5168768 , -0.4077718 ]],
                  [[-2.0171573 ,  0.38832152]],
                  [[-1.4002295 , -0.1987121 ]]], dtype=float32)}}
```

Note that *sigma_t* is not considered a free parameter in the *BinnedAnalysis* class, which is why it still only has one value (for more on this see the `BinnedAnalysis` class documentation).

Also note that we cannot fit to all the samples simultaneously, i.e. we don't use each sample as independent information all contributing simultaneously to knowledge of the underlying parameters. JMCTF is designed for performing Monte Carlo simulations of scientific experiments that run just once (such as a search for new particles at the Large Hadron Collider), so each sample is treated as pseudodata whose main purpose is to help us understand the distributions of test statistics. In this view each sample is an independent pseudo-experiment. If an experiment is in fact to be run multiple times in reality, then the PDF of the underlying Analysis class needs to reflect this by using random variables of the appropriate dimension; or for example by using two normal random variables rather than one if the experiment runs twice.

But back to the example. What we did here was a little weird; we sampled from a distribution that was itself fit to some other samples. More usually, we would sample from a distribution with parameters fixed to some "null hypothesis" (or "alternate hypothesis") values, based on some theoretical motivation. To do this, we can either create the original JointDistribution object with fixed parameters, or fix them in an existing JointDistribution object. But to do this, we need to understand the parameter structure expected by our object. This can be introspected using the `get_parameter_structure` method:

```
>>> free, fixed, nuis = joint.get_parameter_structure()
>>> print("free:", free)
>>> print("fixed:", fixed)
>>> print("nuis:", nuis)
free: {'Test normal': {'mu': 1}, 'Test binned': {'s': 2}}
fixed: {'Test normal': {'sigma_t': 1}, 'Test binned': {}}
nuis {'Test normal': {'theta': 1}, 'Test binned': {'theta': 2}}
```

where 'free' are the free parameters in each analysis that can be fit to data, 'fixed' are non-fittable parameters that must be chosen by the user (or by theory), and 'nuis' are nuisance parameters that enter into the fit but which don't need to be specified when fixing parameters for sampling ("nominal" values for them can be automatically chosen internally).

To fix some "null hypothesis" input parameters for the analyses we need to provide the parameters from the "free" and "fixed" dictionaries above, but not the "nuis" parameters. For example:

```
>>> null = {'Test normal': {'mu': [0.], 'sigma_t': [1.]}, 'Test binned': {'s': [(0.,␣
↪0.)]}}
>>> joint_null = joint.fix_parameters(null)
```

We can then sample from this null distribution and fit the free parameters:

```
>>> samples = joint_null.sample(1e6)
>>> q_full, joint_fitted_full, par_dicts_full = joint.fit_all(samples,null)
>>> q_nuis, joint_fitted_nuis, par_dicts_nuis = joint.fit_nuisance(samples, null)
```

As before (i.e. in the *Combine independent "analyses" into a joint distribution* section), we use the `fit_all` method of our *JointDistribution* object to fit all the free parameters to our samples. Last time we didn't use the second argument, but it is used to supply a parameter dictionary of the "fixed" parameter values to use during the fitting, if any exist. Here we just gave it the full "null hypothesis" parameter dictionary, but all the parameters aside from the fixed parameter 'sigma_t' were ignored.

After this, we used the `fit_nuisance` method to fit *only* the nuisance parameters in the analyses to data, with all the other "free" parameters fixed to values specified in the parameter dictionary given in the second argument. So here the "null hypothesis" values of 'mu' and 's' were taken as fixed, while both 'theta' parameters have been fit.

In the next section, *Find MLEs and conduct simple likelihood ratio tests*, we will see how the results of these fits can be understood in terms of likelihood ratio tests.

3.3 Find MLEs and conduct simple likelihood ratio tests

3.3.1 Likelihood ratio tests

In the previous section (*Sample from the joint distribution*) we learned how to sample from a given `JointDistribution` under a fixed set of null hypothesis parameter values, and how to fit all the free and nuisance parameters for that joint distribution to the simulated data. Now, we investigate the results of those fits and use them to construct likelihood ratio test statistics.

Recall that we had the following fit results:

```
>>> q_full, joint_fitted_full, par_dicts_full = joint.fit_all(samples,null)
>>> q_nuis, joint_fitted_nuis, par_dicts_nuis = joint.fit_nuisance(samples, null)
```

Here, *par_dicts_full* and *par_dicts_nuis* contain the free and nuisance, and nuisance-only, parameters fitted to the same set of *samples*, with non-nuisance free parameters set to null hypothesis values in the latter case. By *fitted* I mean that we have obtained maximum likelihood estimators (MLEs) for those parameters under the simulated samples. I will not explain the full theory of likelihood ratio tests here — for a basic overview the wikipedia page is not bad — but the reason we have done these particular fits is because the combination *q_nuis - q_full* corresponds to a standard type of profile likelihood ratio test statistic and asymptotically follows a chi-squared distribution (when the free parameters in the nuisance-only fit are set to the values used to simulate the samples, i.e. their "true" values). This can be demonstrated with a simple plot:

```
LLR = q_nuis - q_full # -2*log( L_nuis / L_full )
DOF = 3

import matplotlib.pyplot as plt
import seaborn as sns
from tensorflow_probability import distributions as tfd
fig = plt.figure(figsize=(5,3))
ax = fig.add_subplot(111)
ax.set_xlabel("LLR")
ax.set(yscale="log")
sns.distplot(LLR, color='b', kde=False, ax=ax, norm_hist=True, label="JMCTF")
q = np.linspace(0, np.max(LLR),1000)
chi2 = tf.math.exp(tfd.Chi2(df=DOF).log_prob(q))
ax.plot(q,chi2,color='b',lw=2,label="chi^2 (DOF={0})".format(DOF))
ax.legend(loc=1, frameon=False, framealpha=0, prop={'size':10}, ncol=1)
fig.tight_layout()
plt.show()
```

A helper routine is included to simplify making plots like these:

```
from JMCTF.plotting import plot_chi2
fig = plt.figure(figsize=(5,3))
ax = fig.add_subplot(111)
plot_chi2(ax,LLR,DOF)
ax.legend(loc=1, frameon=False, framealpha=0, prop={'size':10}, ncol=1)
fig.tight_layout()
plt.show()
```

which produces the same plot as above.

Knowing the distribution of a test allows one to go on and compute such things as p-values to try and exclude the chosen null hypothesis. Of course, when the test statistic distribution is known then one does not need to do Monte

Carlo simulations, so the usefulness of JMCTF is only manifest when one either cannot analytically determine test statistic distributions at all, or when asymptotic assumptions are suspected or known to fail. We will look at these sorts of issues in section (TODO).

3.3.2 Maximum likelihood estimators

The profile likelihood ratio results in the previous section rely on JMCTF having found accurate maximum likelihood estimates (MLEs) for free parameters. If you are concerned that these are not being correctly found by the minimisation routines, or if you just want to see the values for yourself, then you can simply inspect the parameter dictionaries that are returned from the fitting routines (*par_dicts_full* and *par_dicts_nuis* in the example above). Their structure was discussed at the end of section *Combine independent "analyses" into a joint distribution*. Some helper routines exist to make it easy to plot the simulated distributions of these MLEs (as well as the distributions of the samples):

```
from JMCTF.plotting import plot_sample_dist
fig, ax_dict = plot_sample_dist(samples)
fig.tight_layout()
```

```
from JMCTF.plotting import plot_MLE_dist
fig, ax_dict = plot_MLE_dist(par_dicts_full["fitted"])
# Overlay nuisance-only MLE dists onto full fit MLE dists
plot_MLE_dist(par_dicts_nuis["fitted"],ax_dict)
fig.tight_layout()
plt.show()
```

These plots are currently a bit rough, having only really been used for internal debugging purposes. But you may find them useful for quickly visualising the output of the simulations.

CHAPTER 4

Advanced usage

In these sections you can find explanations and demonstrations on advanced usage of JMCTF, such as adding custom Analysis classes.

4.1 Extending JMCTF with new analysis objects

CHAPTER 5

Look-elsewhere effect

One of the main motivations for creating JMCTF was to facilitate the computation of look-elsewhere effect (LEE) corrections (i.e. corrections for multiple comparisons, or multiple tests). In these sections you will find guidance on the tools available for performing these corrections, along with associated tools for managing large number of simulations/fits for multiple fixed "alternate" hypotheses. In fact you may find the management and database tools useful even if you are not interested in the LEE.

5.1 Background theory and motivation

A full description of the techniques used here can be found in <paper to come>, however I will give a brief overview here. As in the rest of JMCTF, we are primarily concerned with performing hypothesis tests based on likelihood ratios, such as

$$q_\Lambda = -2\log\frac{L(\theta_0,\hat{\eta};x)}{L(\hat{\hat{\theta}},\hat{\hat{\eta}};x)} \tag{5.1}$$

where $L(\theta,\eta;x)$ is a likelihood function parameterised by interesting parameters θ and nuisance parameters *eta*. x is a fixed observation sampled from an associated pdf $p(x|\theta',\eta')$, where θ' and η' are therefore the "true" values of θ and η. In the numerator, $\hat{\eta}$ is the maximum likelihood estimator (MLE) for η with θ fixed to the value θ_0, whilst in the denominator $\hat{\hat{\theta}}$ and $\hat{\hat{\eta}}$ denote the "global" MLE point in the joint $\{\theta,\eta\}$ parameter space.

Under appropriate regularity and smoothness conditions (in accordance with Wilks' theorem), q_Λ asymptotically follows a χ^2 distribution, with degrees of freedom equal to the dimension of θ.

So far so good. In simple cases where you nonetheless worry that asympototic conditions may fail, you can use JMCTF to simulate this test statistic by creating the appropriate joint distribution and doing the required parameter fits. However, you can only do this if the parameters you are interested in are those that directly parameterise your joint distribution.

In more complicated cases, it is common to have some "master theory" that maps onto only a subset of the full parameter space of the direct joint distribution parameters. In the above example, suppose that we have a theory T that acts as the map $T:\phi\rightarrow\theta$. This mapping can in general be arbitrarily complicated, map between very different dimensions of parameter space, and need not be surjective or injective. This mapping may also not be possible to describe via TensorFlow operations, in which case JMCTF has no hope of fitting these parameters for you[1]. To be explicit, when we are interested in testing this sort of "master theory" we can rewrite Eq. (5.1) as

$$q_\Lambda = -2 \log \frac{L(T(\phi_0), \hat{\eta}; x)}{L(T(\hat{\hat{\phi}}), \hat{\hat{\eta}}; x)} \tag{5.2}$$

but the fit in the denominator is not something that JMCTF can do for you.

So, how can we proceed? Suppose that you can provide the mapping T for any input parameters ϕ. In that case, JMCTF can still be used to evaluate test statistics of the form:

$$q_\Lambda = -2 \log \frac{L(T(\phi_0), \hat{\eta}; x)}{L(T(\phi_1), \hat{\hat{\eta}}; x)} \tag{5.3}$$

where we now have a single fixed "alternate hypothesis" $T(\phi_1)$. This can be used to conduct a test of $T(\phi_0)$ vs $T(\phi_1)$, however this is not what we really want to do. We want to perform a test with the whole theory T as the alternate hypothesis, not just fixed points in the parameter space of T.

We can get closer to the desired test statistic by doing *many* tests of the kind in Eq. (5.3), however if we do something like choose the lowest p-value out of these tests as our reported p-value for excluding $T(\phi_0)$ then we will have done a kind a "p-hacking", i.e. performed "multiple comparisons", i.e. fallen victim of the Look-elsewhere effect. However, this means that our problem can be re-imagined as one of correcting for this look-elsewhere effect.

To do this, rather than conduct many separate tests, we want to effectively combine them all into one test. So, suppose we have a list of "alternate hypotheses" from the parameter space of T that "adequately"[2] spans the space ϕ. For each trial observation x, then, rather than find $\hat{\hat{\phi}}$ we find the alternate hypothesis that gives the highest likelihood out of the candidate set. Let this hypothesis be labelled $\tilde{\phi}$. Our test statistic can then be written as

$$q_\Lambda = -2 \log \frac{L(T(\phi_0), \hat{\eta}; x)}{L(T(\tilde{\phi}), \hat{\hat{\eta}}; x)} \tag{5.4}$$

Given a suitably comprehensive set of alternate hypotheses, then, this test should give results close to those of Eq. (5.2). Unfortunately this remains a rather computationally expensive process: for each of (say) 10^6 simulated observations x, we need to fit the nuisance parameters for (say) $10^3 - 10^6$ (or more) alternative hypotheses, and then select the one that gives the highest likelihood. This is an awful lot of fitting, even for the fast gradient descent fitters of TensorFlow.

JMCTF provides tools to perform and manage all these fits, and also includes routines for making some extra approximations to speed things up. These are described further in section TODO.

5.2 A simple example of a look-elsewhere effect correction

[1] In fact JMCTF cannot currently help you even if the mapping *can* be represented in TensorFlow operations. This would be quite a powerful feature, and it could be added in the future if there is demand for it, but for now it was not necessary for the projects that motivated the creation of JMCTF.

[2] The issue of what constitutes an "adequate" set of fixed alternate hypotheses from T is not an easy one to answer, however we can say that ideally one should provide enough alternate hypotheses so that, for any sample observation x, the likelihood value under the "best fit" fixed alternate hypothesis is sufficiently close to the likelihood value under the "true" MLE of ϕ.

CHAPTER 6

Examples from quick start

Return to *Quick Start* guide.

CHAPTER 7

jmctf package

7.1 Module contents

- `NormalAnalysis`
- `BinnedAnalysis`

7.2 Analysis classes

7.2.1 NormalAnalysis

7.2.2 BinnedAnalysis

7.3 JointDistribution

7.4 jmctf.common module

CHAPTER 8

Indices and tables

- genindex
- modindex
- search