Google Summer of Code 2019 Proposal

Topic: Probability

Student: Ritesh Kumar

About Me

Name: Ritesh Kumar

University: Indian Institute of Technology Kanpur

Program: BTech Computer Science and Engineering & BTech Electrical Engineering

Contact Information

Github: ritesh99rakesh

E-mail: ritesh99rakesh@gmail.com, riteshk@cse.iitk.ac.in

Phone Number: (+91) 7318015400

Timezone: Indian Standard Time(UTC +05:30)

Programming Setup

OS: Ubuntu 16.04

Editor: Atom

Version Control: Git

Personal Background

- Academic Experience

I am a third-year undergraduate student at the Indian Institute of Technology Kanpur. I am pursuing BTech in Electrical Engineering with the second major in Computer Science and Engineering. My current CPI is 9.8/10 after five semesters. Few of my courses relevant for the project:

  - Probability and Statistics
  - Randomized Algorithms
  - Algorithms - I and II
  - Probabilistic Machine Learning and Inference
  - Linear Algebra and Complex Analysis

- Programming Experience

  I am proficient in C, C++, Python, Julia, Javascript. My field of interest includes Randomized Algorithms, Probabilistic Machine Learning and Robotics.

  I am accustomed to C++ and Object Oriented Programming. I was introduced to Python three years ago, and since then I have used it in multiple projects both course-related and otherwise. I am reasonably comfortable with Git and Github. I have also become quite comfortable with SymPy as I have been using and contributing to it for past months.

The Project

Introduction

My target of GSoC'19 under SymPy would be to develop the stats module to increase its functionality like

- robust multivariate distributions
- compound distribution
- ability to export expressions of RVs to external libraries
- support integration and equation solving and
- introduce new features like Markov Chain, Random Walk and Random Matrices.

Motivation

My motivation towards the project is my interest in probability and statistics, and algorithms. Also, due to my exposure to (probabilistic) machine learning, I feel that SymPy in general and Stats module, in particular, could be handy in this and other fields. Moreover, the people involved with SymPy has been so welcoming and have inspired to take up this project.

Implementation plans

STAGE 1

I plan to start the project by implementing the following distributions:

- Univariate(Continuous):
  - Inverse Gaussian distribution
  - Levy distribution
  - Non-Central Beta distribution
  - Beta-Binomial distribution
  - Poisson Binomial distribution
- Multivariate(Continuous):
  - Wishart distribution
  - Inverse Wishart distribution
  - Dirichlet distribution
  - Inverted Dirichlet distribution
  - Multivariate Pareto distribution
  - Normal Inverse Gamma distribution
  - Normal Inverse Wishart distribution
  - Normal Wishart distribution
- Multivariate(Discrete):
  - Multinomial distribution
  - Negative Multinomial distribution

I also plan to write a test to all the distributions above and increase the current code coverage(I am currently working on these).

STAGE 2

Compound Distributions

SymPy allows for the creation of compound RVs. The density can be calculated for these variables but, the probability of an event can't be calculated:

```
>>> x = Symbol('x')
>>> N = Normal('N', 0, 1)
>>> M = Normal('M', N, 1)
>>> simplify(density(M)(x).doit())
exp(-x**2/4)/(2*sqrt(pi))
>>> P(M > 2)
RuntimeError: maximum recursion depth exceeded while calling a Python object
```

At present, if a given expression contains compound RVs, the probability space is taken as a `JointPSpace`:

```
if any(isinstance(arg, RandomSymbol) for arg in args):
    pspace = JointPSpace(symbol, CompoundDistribution(dist))
```

(stats/crv_types.py and stats/drv_types.py)

Which assumes that the RV belongs to `JointPSpace`. This concept mixes both joint and compound distributions and complicates the further computation of compound RV.

To solve this issue, I plan to implement compound RVs in a separate file(`compound_rv.py`), Very similar to joint distribution. This also requires to implement a separate class of probability space called `CompoundPSpace`. An object of this class will be created if the parameters of the given RV are found to be distributed according to another RV.

For example, while creating a compound continuous RV:

```
def compound_rv(cls, sym, *args):
    args = list(map(sympify, args))
    dist = cls(*args)
    args = dist.args
    dist.check(*args)
    return CompoundPSpace(sym, dist).value
```

(similar to `multivariate_rv()` for `joint_rv`)

I couldn't find a general method to compute the density of compound RV *(I believe that such general formula doesn't exist, because it would require integrating out at least one parameter, e.g. non-conjugate prior in machine learning)*. But for some instances, it is possible as given here.

Consider the following code of how this case can be dealt with:

```
class CompoundPSpace(PSpace):
    def __new__(cls, s, distribution):
        dist = compute_compound_type(distribution)
        return Basic.__new__(cls, s, dist)
```

`compute_compound_type` is a function to find the resulting distribution after compounding. This can be done in stages:

1. Find out what my original distribution is. Say, `mu ~ Normal('mu', mu0, sigma0**2)`.
2. Find out how the distribution is compounded, i.e. which parameter is compounded and which distribution it is. Say `X ~ Normal('X', mu, sigma**2)`
3. Return the resultant distribution with the modified parameters. In our case, it is `Normal`, `X ~ Normal('X', mu0, sigma0**2 + sigma**2)`

Code for above example would like:

```
def compute_compound_type(distribution, **kwargs):
    ...
    if isinstance(distribution, NormalDistribution):
        mean, var = distribution.args()
        if isinstance(mean, NormalDistribution) and not \
            isinstance(var, RandomVariable):
            dens = density(mean)
            return NormalDistribution(dens.args[0], var + dens.args[1])
```

An unevaluated object would be returned if the type of distribution can't be evaluated.

Compound distributions are very common in Machine Learning especially Probabilistic ML. A well known example from Probabilistic ML is that of Negative Binomial distribution(NBD). The NBD arises as a continuous mixture of Poisson distributions (i.e. a compound probability distribution) where the mixing distribution of the Poisson rate is a gamma distribution.

```
>>> k = Symbol("k", positive=True)
>>> theta = Symbol("theta", positive=True)
>>> lamda = Gamma("lamda", k, theta)
>>> X = Poisson("x", lamda)
>>> # therefore above function would return
>>> compute_compuound_type(X)
NegativeBinomial("x", k, theta/(theta+1))
```

STAGE 3

Stochastic Process

A stochastic or random process can be defined as a collection of RVs that is indexed by some mathematical set.

On this topic, some work was done in GSoC'18, and I would continue it further.

Work was done in GSoC'18:

- `StochasticProcess()` class was created under `joint_rv.py`, which is an abstract class representing stochastic processes as a collection of joint distributions.
- New python file `stochastic_process_types.py` was created and `BernoulliProcess` was added (it was not completed).

I would like to take up the work from where it was left. I will first complete `BernoulliProcess` module along the lines of this comment.

Then I will take up following two important stochastic processes:

1. Markov Chains:

I will create a new class `MarkovProcess/MarkovChain` under `stochastic_process_types.py`. Following concepts are involved [Reference]:

- *Transition Probability Matrix (`transMatrix`):* A square matrix where the rows indicate the current state and column indicate the transition. The rows sum to 1.
- *Initial State Probability (`s0`):* This is the row vector, representing initial probabilities of being in different states. The row sum to one.
- *t state vector (`st`):* `st = s0*transMatrix**t`
- *Stationary vector (`stationary`):* Obtained by solving `s*transMatrix = s`.
- *Existence of Stationary vector:* If a Markov chain is regular, then it has a unique stationary matrix.
- *Absorbing state:* A state in a Markov chain is called an absorbing state if once the state is entered, it is impossible to leave.

- *Absorbing Markov chain:* If there is at least one absorbing state and it is possible to go from any state to at least one absorbing state in a finite number of steps.

  Crude class MarkovChain will be as follows:

```
class MarkovProcess(StochasticProcess):
  def __init__(self, name, transMatrix, s0):
    # checks
    if not isinstance(transMatrix, Matrix):
      raise TypeError()
    for i in transMatrix.shape[0]:
      if sum(transMatrix[i]) != 1:
        raise ValueError()
    if s0 != None and not isinstance(s0, Matrix)
      raise TypeError()
    if isinstance(s0, Matrix) and sum(s0) != 1:
      raise ValueError()
    self.name = sympify(name)
    self.transMatrix = sympify(transMatrix)
    self.s0 = sympify(s0)
  def initial_state(self):
    if s0 != None:
      return self.s0
  def transition_matrix(self):
    return self.transMatrix
  def state(self, t):
    return self.s0*self.transMatrix**t
    # we must speed up this matrix multiplication, as this can be the bottleneck
  def stationary_state(self):
    """ Use Perrson_Frobenius given
    https://en.wikipedia.org/wiki/Perron%E2%80%93Frobenius_theorem
    to check for existence of stationary_state """
    return solve(transMatrix.T - I)
  def is_absorbing(self):
    for i in range(len(self.transMatrix)):
      if self.transMatrix[i][i] == 1
      return True
```

```
            return False
        def canonical_form(self):
            if not self.is_absorbing():
                return ValueError()
            else:
                """ Implement code given
                https://github.com/mkutny/absorbing-markov-chains/blob/master/amc.py
                we get P = [[Q, R], [0, I_r]] where Q is t-by-t matrix (t=number_of_transients),
                R is nonzero t-by-r matrix, 0 is r-by-t zero matrix and
                I_r is r-by-r identity matrix (r=number_of_absorbing_states)
                return canonical_form_matrix """
        def fundamental_matrix(self):
            canonical_matrix = self.canonical_form()
            # Get Q from canonical matrix
            Q = canonical_matrix[Q]
            return (eye(number_of_transients) - Q)**(-1)
        """ Also implement Variance on number of visits,
            Expected number of steps, Variance on number of steps,
            Transient probabilities and Absorbing probabilities as given here:
            https://en.wikipedia.org/wiki/Absorbing_Markov_chain """
```

Example usage:

```
>>> MChain = MarkovProcess("x", [[0.98, 0.02], [0.78, 0.22]], [0.90, 0.10])
>>> MChain.transition_matrix()
[[0.98, 0.02], [0.78, 0.22]]
>>> MChain.state(1)
[0.96, 0.04]
>>> MChain.stationary_state()
[0.975, 0.025]
```

2. Random walks: The aim would be to implement random walks in one, two and three dimensions since these are the ones that are most often used. I plan to implement them in the cartesian space pertaining to the dimension of the walk.

   Each random walk will be implemented as a separate class. I plan to do so because there are many results like the probability of reach a certain coordinate, which have a direct result in a one-dimensional walk but not in a multidimensional walk. I also plan to implement functions that would compute the properties of a given random walk.

   A functioning but quite naive prototype of a random walk in one dimension is given below.

```
class RandomWalk_1D(Basic):
    def __init__(self, p=S.Half, init_position=S.Zero, step_size=1):
        self.p = p
        self.q = S.One - p
        self.step_size = step_size
        self._position = init_position
        self._path = [0]
     def position(self):
         """ Returns current position """
```

```
        return self._position
    def path(self):
        """ Return the path taken to the current position """
        return self._path
    def move(self):
        """ Take a single step """
        if random.random() < self.p:
            self_position += 1
        else:
            self._position -= 1
        self._update_path(self.position())
    def walk(self, n):
        """ Takes a walk in n steps """
        for i in range(n):
            self.move()
    def _update_path(self, n):
        self._path += [n]
    def expected_right(self, steps):
        """ Return number of expected steps
            taken in right direction """
        return self.p*steps
    def expected_left(self, steps):
        """ Return number of expected steps
            taken in left direction """
        return self.q*steps
    def prob_position(self, pos=0, steps):
        """ Probability that we are position=pos
            after N steps starting from position=self._position """
        dist = pos - self._position
        if abs(dist) > steps or (steps + dist) % 2 == 0:
            return 0
        else:
            n1 = (steps + dist)/2
            n2 = (steps - dist)/2
            return Binomial(steps, n1)*self.p**n1*self.q**n2
    def prob_distance(self, dist=1, steps):
        """ Probability that we are at distance=dist
            from position=self._position after N steps """
        return self.prob_position(dist+self._position, steps)
    def moment(self, t, steps):
        """ Returns moment `mu` of prob_distance distribution """
        d = Symbol('d', Integer=True)
        return summation((d**t)*prob_distance(d, steps), (d, -steps, steps))
    def mean(self, steps):
        """ Returns mean """
        return moment(1, steps)
    def variance(self, steps):
        """ Returns variance """
        return moment(2, steps)
    def skewness(self, steps):
        """ Returns skewness """
        return moment(3, steps)
    def kurtosis(self, steps):
        """ Returns kurtosis """
        return moment(4, steps)
```

Example usage:

```
>>> RWalk = RandomWalk_1D()
>>> RWalk.move()
RWalk.position()
1
>>> RWalk.walk(steps=5)
>>> RWalk.path()
[0, 1, 2, 1, 0, 1, 0]
>>> RWalk.expected_right(steps=10)
5
>>> RWalk.prob_position(pos=3, steps=2)
0
```

I also plan to implement Gaussian Random walk, though this will not be top priority as any new walk, could now be added along the same lines as above and I aim to complete `SimpleRandomWalk` for one, two and three dimensions in its entirity and then move ahead with other random walks.

STAGE 4

Random Matrices

With all the distributions implemented and with the multivariate distribution at par with univariate counterpart, the next step would be to implement Random matrices.

I will write two files namely `random_matrix.py` and `random_matrix_types.py`. `random_matrix.py` will be along the lines of `crv.py` and I will define the `MatrixPSpace` for random matrices. In `random_matrix_types.py`, I plan to implement various types of random matrices distribution and follow a similar layout to `crv_types.py`. Few of them are `MatrixNormalDistribution` and `MatrixTDistribution`.

I first plan to implement `MatrixNormalDistribution`. For this, I will reference from Wolfram. I will also add/update functions to calculate the mean, variance and kurtosis of this distribution.

Expected working;

```
>>> matrixNormal = MatrixNormalDistribution('N', mu=\
[[1, 4], [6, 8], [10, 2]}, sigma_row=diag(2, 1, 3),\
sigma_col=[[2, 1], [1, 3])
>>> mean(matrixNormal)
[[1, 2], [6, 8], [10, 2]]
```

I have not thought the complete implementation details for `MatrixRandomDistribution` and the community bonding period will be a good time for discussion of ideas with the mentor.

STAGE 5

Export expressions of RV to external libraries(PyMC3, PyStan)

I have used PyMC3 in my Machine Learning courses. Since both are python libraries, exporting to these libraries would deal with translating SymPy RVs so that they can be fed into them.

Below is an example, to convert SymPy Normal RV to PyMC3 RV:

```
from sympy.stats import density
import pymc3 as pm
def convert_normal_pymc3(normal_rv):
    """
    Given parameters of sympy Normal RV, PyMC3 RV is returned
    """
    name = str(normal_rv.args[0])
    mean, std = (int(arg) for arg in density(normal_rv).args)
    with pm.Model():
        pm_normal = pm.Normal(name, mean, std)
    return pm_normal
```

Dependence between RV

Current state on measures of dependence between RV:

- Pearson Coefficient and Covariance implemented
- Following works

```
>>> from sympy.stats import Normal, Exponential, correlation
>>> E = Exponential('E', 1)
>>> N = Normal('N', 0, 1)
>>> X = Normal('X', [1, 2], [[1, 0], [0, 1]])
>>> correlation(E, E+N)
sqrt(2)/2
>>> correlation(X[0], X[1])
0
```

  - The following does not work

```
>>> from sympy.stats import Normal, Exponential, correlation
>>> E = Exponential('E', 1)
>>> N = Normal('N', 0, 1)
>>> M = Normal('M', N, 1)
>>> correlation(E, M)
NotImplementedError
>>> correlation(N, M)
NotImplementedError
```

    With the implementation of compound probability distributions and current implementation of dependency measures, I think that this should not be tough. I will discuss more about this with the mentor.

STAGE 6 (Stretch Goal)

Support integration and equation solving

I will work to implement functions like quantile, median, mode. Also, this is not complete, and I would discuss with the mentor during community bonding time about what can be done in this section.

Timeline

Community Bonding Period

Goals for this period are:

1. Get to know my mentor and other organizations members well.
2. Get the idea of workflow during the project.
3. Implement the distributions mentioned in stage 1 and add tests for them.
4. Discuss with mentors about the implementation details of Random Matrix.
5. Read more about Random walk in two and three dimensions from *Principles of Random Walk by Spitzer*.

Coding Period

May 27 - June 2 (Week 1)

- I shall work on compound distributions. Refactor current `joint_rv.py` to include the suggested changes.
- I will implement `CompoundPSpace`.
- Update documentation of community bonding period work and complete any remaining work.
- Completion of Stage 1.

June 3 - June 16 (Week 2 and 3)

- I will implement a function to find distributions for some compound distributions as suggested above.
- I will complete `BernoulliProcess` and implement `MarkovChain` and write tests for it.
- I will implement Random Walk in one dimension.
- I will also study more about Random Walk in two and three dimensions.
- Completion of Stage 2.

June 17 - June 23 (Week 4)

- I will implement Random Walk in two dimensions.
- I will also research about the implementation of Gaussian Random Walk.

June 24 - June 30 (Week 5)

- Buffer Period: Catch up with any unfinished work left in the past weeks.
- Preparation for Phase 1 Evaluation Submission Deadline which is on June 28.
- I will also write a blog post describing my work and experience so far.

July 1 - July 14 (Week 6 and 7)

- I will implement Random Walk in three dimensions.
- I will also implement Gaussian Random Walk in one dimension.
- Discuss with mentors about implementation details of Random Matrices.
- Will also study the implementation given in Wolfram.
- Completion of Stage 3.

July 15 - July 21 (Week 8)

- Start with the basic structure for Random matrices.
- Will write code for probability space of Random matrices.

July 22 - July 28 (Week 9)

- Buffer Period: Catch up with any unfinished work left in the past weeks.
- Preparation for Phase 2 Evaluation Submission Deadline which is on July 26.
- Write a blog post providing updates on my work and experience working doing this project.

July 28 - August 10 (Week 10 and 11)

- I will implement `MatrixNormalDistribution`, add various functions like mean, variance, skewness to it and write tests for them.
- I will also write code for exporting SymPy RVs to PyMC3 and PyStan.
- Discussion about supporting the assumption of dependence between RVs.
- Completion of Stage 4.

August 11 - August 17 (Week 12)

- Complete the export functions.
- Write code for support of dependence between RVs.
- Completion of Stage 5.

August 18 - August 26 (Week 13 and Wrap up)

- Preparation for Final Phase Evaluation Submission Deadline which is on August 26. This includes code-formatting, completing documentation and solving unknown bugs.
- I will write a blog post recounting my experience and work during the whole project.
- If time is left, then I can work on the implementation of functions like quantile, median, mode and supporting integration.

Why Me?

I am proficient in writing code using Python. Due to my exposure to sympy and coursework on Probability and Statistics, Probabilistic Machine Learning, I am quite familiar with solving complex probability problems and writing code in SymPy. My PR to SymPy are:

Merged

- #16338 Physics.optics: Added tests to physics.optics for increased coverage.
- #16440: Physics.mechanics: Added tests for particle and rigidbody in physics.mechanics.
- #16449 stats: Added CDF for Maxwell, Rayleigh, Cauchy and Gompertz distributions.

Open

- #16152 Printing: Corrected printing symbol which contained text.
- #16202 Physics.mechanics: Added functions to find rotational_kinetic_energy, translational_kinetic_energy and total_kinetic_energy for rigidboy and particle.
- #16361 stats: Added kurtosis function and test for it.
- #16465 stats: Added Levy distribution to crv_types.py and added tests for it.

Availablity and Working Hours

I have no other commitments during the summer. My holiday's duration match with that of GSoC. During this time I will be able to devote 40+ (minimum) hours per week. My college reopens in August. But due to little load during the beginning, I still would be able to devote 40+ hours.

I do not have particular working hours and can adjust my routine according to mentor, to achieve maximum efficiency. I will always be available for replying to messages.

References

- SymPy Docs
- Wikipedia and Wolfram mathworld
- Principles of Random Walk by Spitzer F.
- Markov Chains by Norris J.
- Past year proposals
