projectpredict Documentation

Release 0.1

Justin Tervala

May 15, 2018

Contents

Welcome to the documentation for ProjectPredict, the library to project managers schedule tasks intelligently. Just getting started? Read the *What is ProjectPredict?* section. Interested? Read the *Installation* section to get ProjectPredict and get started.

CHAPTER 1

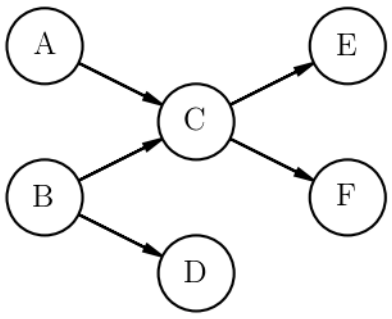
What is ProjectPredict?

ProjectPredict is a library to help project managers gain insight into the status of their project using Bayesian networks. It is inspired by the paper "Project scheduling: Improved approach to incorporate uncertainty using Bayesian networks" (Khodakarami, Fenton, & Neil, Project Management Journal, 2007). The project features

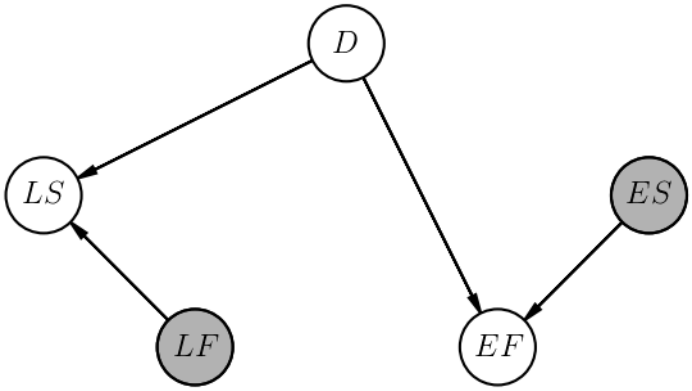
- Inferring the latest start date, earliest finish date, and total float for each task in a project
- Recommending which task or tasks should be started next using custom constraints and objective functions
- Task duration specified either through three-point (PERT) estimation or inferring the duration of a task from a machine learning model
- Visualization of a project timeline using Matplotlib

1.1 The Bayesian network

A project is specified as a directed acyclic graph of tasks. For example, suppose you have three tasks, A, B, C, D, E, and F. Task C can only be begun when tasks A and B are completed, task D can only be completed when task B is completed, and tasks D and E can only be begun when task B is completed. The resulting graph would look like this:



Each task is then decomposed into a smaller Bayesian network.



Where D is the duration, ES is the earliest start date, LS is the latest start date, EF is the earliest finish date, and LF is the deadline or latest finish date. The earliest finish date can be inferred from the graph by traversing the graph in topological order from the starting tasks (A and B in our example), from the equations

$ES_i = \max\{ES_j + D_j \ \forall \text{ predecessor tasks } j\}$

$EF_i = ES_i + D_i$

The latest start date for each task can be inferred by traversing teh graph in reverse topological order from the final tasks (D and E in our example), from teh equations

$LF_i = \max\{LF_j - D_j \ \forall \text{ successor tasks } j\}$

$LS_i = LF_i - D_i$

For our sample project, tasks A and B must be given an earliest start date, and tasks C and D must be given a latest finish date. Both of these can take the form of either a probability distribution or a hard date. All tasks must be given a duration, either using three-point estimation or predicted from a learning model.

Once these values have been inferred for each task, the total float can be defined as $TF_i = LF_i - EF_i$. This is a measure of the amount of time a task's duration can be increased without affecting the completion time of the project as a whole. The smaller the total float of a task, the more critical the task is to the overall project.

CHAPTER 2

Installation

The easiest way to install ProjectPredict is to install it from PyPI using pip

```
pip install projectpredict
```

Or, using Pipenv, the new officially recommended standard for Python package management,

2.1 Development Installation

Currently the only way to install ProjectPredict for development is to clone it from GitHub.

```
git clone https://github.com/JustinTervala/ProjectPredict
```

Set up your virtual environment using virtualenv

```
git clone https://github.com/JustinTervala/ProjectPredict
cd ProjectPredict
virtualenv venv
source venv/bin/activate
```

Then install the requirements

```
pip install -r requirements.txt
pip install -r requirements-dev.txt
```

Or, using Pipenv

```
git clone https://github.com/JustinTervala/ProjectPredict
cd ProjectPredict
pipenv install --dev
pipenv shell
```

2.2 Testing

ProjectPredict uses pytest as its unit testing framework. You can run the tests from the top-level directory by simply typing "pytest"

```
pytest --cov=projectpredict
```

2.3 Building the Documentation

ProjectPredict uses sphinx to build the docs, and uses several plugins. From the top-level directory,

```
cd docs
pip install -r requirements.txt
make html
```

This will generate the file in docs/_build/index.html. This file is the entry point to the documentation

CHAPTER 3

The Recommendation Engine

ProjectPredict comes with a flexible recommendation engine which can be used to determine which tasks should be started next. You can constrain the set of tasks both by a minimum and maximum number of tasks as well as by using custom constraint functions. You can also specify if all tasks must be completed before the next tasks can begin or if a new set or tasks can be started whenever any of the tasks in the recommended set completes. The default algorithm selects a set of tasks which maximizes the sum of the total float across the project, weighted by the importance of some tasks' deadlines and the risk tolerance.

3.1 The Default Algorithm

The default algorithm iterates through all possible combinations of tasks which can be started (all tasks with no uncompleted predecessors) and, for each combination infers the latest start date, earliest finish date, and total float of each task in the project assuming that the combination of tasks is begun at the current time. For each combination it creates two scores, the float score and the precision score as defined by

$s_f = \sum_{\text{tasks } i} w_i \mu_i$

$s_p = \sum_{\text{tasks } i} w_i / \sigma_i$

Where μ_i is the mean total float for task i, σ_i is the mean total float for task i, and w_i is the weight of the deadline for task i (defaults to 1 if unspecified).

These scores are then used to select the best combination of tasks. First each score is scaled linearly between 0 and 1 based on the minimum and maximum of both scores.

$\bar{s_f} = \frac{s_f - \min_{\text{task set i}} s_{f_i}}{\max_{\text{task set i}} s_{f_i}}$

$\bar{s_p} = \frac{s_p - \min_{\text{task set i}} s_{p_i}}{\max_{\text{task set i}} s_{p_i}}$

Where $\bar{s_f}$ and $\bar{s_p}$ are the scaled total float score and scaled precision respectively for a task. These two are then combined with a risk tolerance factor, r, a value from 0 to 1, to obtain the combined score s, using $s = r\bar{s_f} + (1-r)\bar{s_p}$. The recommended task set is the set of tasks which has the maximum combined score.

3.2 Customization

The recommendation algorithm can be customized by specifying a scoring function which will accept the earliest start date, latest start date, earliest finish date, latest finish date, and total float samples generated for a task set as well as some optional keyword arguments. A recommendation selection function must also be supplied which accepts the generated scores and some optional keyword arguments. A list of constraints can be specified by supplying a list of functions which accept the project and a proposed set of tasks and returns a boolean indicating if the set of task satisfies the constraints. For examples see *Recommendations with Constraints*

CHAPTER 4

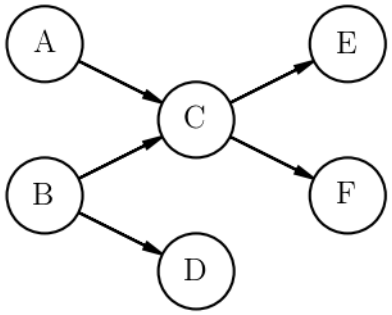
Examples

4.1 Your First Project

The simplest way to construct a project is to use deterministic distributions for the duration, earliest start date, and latest start date. Suppose our project has 6 tasks – A, B, C, D, E, F specified as

Task	Duration	Earliest start date	Latest finish date
A	1 day	Anytime	–
B	3.5 hours	2018-05-14 12pm	–
C	2 days	–	–
D	3 days	–	2018-04-16
E	1 hour	–	2018-05-15
F	5 hours	–	2018-05-20

With the following dependencies



We first write create Tasks from DurationPdfs for the durations and DatePdfs for the earliest start and latest finish dates

```python
from datetime import datetime

from projectpredict import Project, Task, TimeUnits, DurationPdf, DatePdf
from projectpredict.pdf import DeterministicPdf


taskB_earliest_start_date = datetime(year=2018, month=5, day=14, hour=12)

# We make a DatePdf centered around taskB_earliest_start_date.
# The second parameter should be a zero-mean distribution.
# Because this start date is fully deterministic, we use a DeterministicPdf
# with value of 0
taskB_earliest_start_pdf = DatePdf(taskB_earliest_start_date, DeterministicPdf(0))

#

# Because Task A doesn't specify an earliest start date pdf it is assumed that
# it can begin any time.
taskA = Task(
    'A',
    duration_pdf=DurationPdf(DeterministicPdf(1), units=TimeUnits.days)
)

taskB = Task(
    'B',
    duration_pdf=DurationPdf(DeterministicPdf(3.5), TimeUnits.hours),
    earliest_start_date_pdf=taskB_earliest_start_pdf
)

taskC = Task(
    'C',
    duration_pdf=DurationPdf(DeterministicPdf(2), units=TimeUnits.days)
)


# Final tasks require a latest finish date
taskD_latest_finish_date = datetime(year=2018, month=5, day=16)
taskE_latest_finish_date = datetime(year=2018, month=5, day=15)
taskF_latest_finish_date = datetime(year=2018, month=5, day=20)


taskD = Task(
    'D',
    duration_pdf=DurationPdf(DeterministicPdf(3), units=TimeUnits.days),
    latest_finish_date_pdf=DatePdf(taskD_latest_finish_date, DeterministicPdf(0))
)

taskE = Task(
    'E',
    duration_pdf=DurationPdf(DeterministicPdf(1), units=TimeUnits.hours),
    latest_finish_date_pdf=DatePdf(taskE_latest_finish_date, DeterministicPdf(0))
)

taskF = Task(
    'F',
    duration_pdf=DurationPdf(DeterministicPdf(5), units=TimeUnits.hours),
    latest_finish_date_pdf=DatePdf(taskF_latest_finish_date, DeterministicPdf(0))
)
```

Once we have defined the tasks, we can add the tasks and their dependencies to the project.

```
# Construct a Project with the name "MyProject"
project = Project('MyProject')

tasks = [taskA, taskB, taskC, taskD, taskE, taskF]
dependencies = [
    (taskA, taskC),
    (taskB, taskC),
    (taskB, taskD),
    (taskC, taskE),
    (taskC, taskF)
]
project.add_tasks(tasks)
project.add_dependencies(dependencies)
```

Finally we can get the derived latest start date, earliest finish date, and total float for the tasks.

```
# We can specify a current time. If not specified, then
# The current wall time is used
current_time = datetime(year=2018, month=5, day=12, hour=12)

# Because all the distributions are deterministic, we only need 1 iteration
stats = project.calculate_task_statistics(current_time=current_time, iterations=1)

taskA_stats = stats[taskA]

print('earliest finish: {}'.format(taskA_stats.earliest_finish))
print('latest start: {}'.format(taskA_stats.latest_start))
print('total float: {}'.format(taskA_stats.total_float))
```

```
"earliest finish: {'variance': datetime.timedelta(0), 'mean': datetime.datetime(2018,␣
↪5, 13, 12, 0)}"
"latest start: {'variance': datetime.timedelta(0), 'mean': datetime.datetime(2018, 5,␣
↪11, 23, 0)}"
"total float: {'variance': datetime.timedelta(0), 'mean': datetime.timedelta(-1,␣
↪39600)}"
```

For this particular project, the total float is negative, indicating that Task A appears to already be past the deadline. Additionally, we could use calculate_earliest_finish_times() and calculate_latest_start_times() methods to calculate only the earliest finish dates and latest start dates respectively.

4.2 Using Distributions

The world is almost never kind enough to let us know the exact duration of a task, and some deadlines are more flexible than others, and some earliest start dates may be uncertain. Rather than blindly guessing a distribution for the durations, we'll use three-point (PERT) estimation to derive the distribution using the Task.from_pert() method.

Task	Duration		
	Best Case	Expected	Worst Case
A	5 hours	24 hours	36 hours
B	0.5 hours	3.5 hours	10 hours
C	1 day	2 days	4 days
D	0.5 days	3 days	7 days
E	0.2 hours	1 hour	4 hours
F	1 hour	5 hours	10 hours

We'll also put a zero-mean Gaussian distribution over the earliest start date of Task B and the latest finish date of Task D.

```
from projectpredict.pdf import GaussianPdf

taskB_earliest_start_date = datetime(year=2018, month=5, day=14, hour=12)

taskA = Task.from_pert('A', 5, 24, 36, units=TimeUnits.hours)

taskB = Task.from_pert('B', 0.5, 3.5, 10, units=TimeUnits.hours,
    earliest_start_date_pdf=DatePdf(
        taskB_earliest_start_date,
        GaussianPdf(0, 2),
        units=TimeUnits.hours)
)

taskC = Task.from_pert('C', 1, 2, 4, units=TimeUnits.days)

taskD_latest_finish_date = datetime(year=2018, month=5, day=16)
taskE_latest_finish_date = datetime(year=2018, month=5, day=15)
taskF_latest_finish_date = datetime(year=2018, month=5, day=20)


taskD = Task.from_pert('D', 0.5, 3, 7, units=TimeUnits.days,
    latest_finish_date_pdf=DatePdf(
        taskD_latest_finish_date,
        GaussianPdf(0, 1),
        units=TimeUnits.days
    )
)

taskE = Task.from_pert('E', 0.2, 1, 4, units=TimeUnits.hours,
    latest_finish_date_pdf=DatePdf(taskE_latest_finish_date, DeterministicPdf(0))
)

taskF = Task.from_pert('F', 1, 5, 10, units=TimeUnits.hours,
    latest_finish_date_pdf=DatePdf(taskF_latest_finish_date, DeterministicPdf(0))
)
```

From here, we can add the tasks and dependencies to a Project and calculate the statistics same as in the previous example.

4.3 Learned Model

While using three-point estimation is much better than either deterministic or guessing a distribution, it would be even better to learn the distribution from a model. Imagine you are using an issue tracker for a software project. Frequently you'll have some knowledge of what team the work will be done by and the story points of the task. You may also have some history of how long each task took to complete. Using this information, you could train a model to determine the duration a task will take. ProjectPredict currently supports using a Gaussian Process Regression model from scikit-learn to predict the duration of the task. We'll first generate some simulated data for the project. We'll assume the durations are in units of days.

```
import numpy as np
from scipy.stats import norm
import pandas as pd


# We give out teams integer keys, a name, and a probability that any given
# task will be assigned to them
teams = {
    1: {'team': 'red', 'prob': 0.5},
    2: {'team': 'blue', 'prob': 0.25},
    3: {'team': 'green', 'prob': 0.15},
    4: {'team': 'yellow', 'prob': 0.1},
}

# For each team (by number), what give the probability that the team will
# assign some points to any task.
team_points = {
    1: [{'points': 1, 'prob': 0.5},
        {'points': 2, 'prob': 0.3},
        {'points': 3, 'prob': 0.2}],

    2: [{'points': 1, 'prob': 0.4},
        {'points': 2, 'prob': 0.4},
        {'points': 3, 'prob': 0.2}],

    3: [{'points': 1, 'prob': 0.7},
        {'points': 2, 'prob': 0.25},
        {'points': 3, 'prob': 0.05}],

    4: [{'points': 1, 'prob': 0.3},
        {'points': 2, 'prob': 0.5},
        {'points': 3, 'prob': 0.2}],
}

# Assign the mean and std of a Guassian distribution to
duration_lookup = {
    1: {1: {'mean': 3, 'std': 0.5},
        2: {'mean': 5, 'std': 1.25},
        3: {'mean': 10, 'std': 2}},

    2: {1: {'mean': 1, 'std': 0.5},
        2: {'mean': 3, 'std': 2},
        3: {'mean': 5, 'std': 3}},

    3: {1: {'mean': 2, 'std': 1},
        2: {'mean': 4, 'std': 3},
        3: {'mean': 7, 'std': 4}},
```

```
    4: {1: {'mean': 1, 'std': 0.5},
        2: {'mean': 2, 'std': 1.15},
        3: {'mean': 4, 'std': 5}},
}


def generate_team_samples(teams, num_samples=100):
    return np.random.choice(
        list(teams.keys()), p=[team['prob'] for team in teams.values()], size=num_
↪samples)


def generate_points_samples(team_points_lookup, team_samples):
    results = []
    for team_sample in team_samples:
        lookup = team_points_lookup[team_sample]
        points = np.random.choice(
            [entry['points'] for entry in lookup],
            p=[entry['prob'] for entry in lookup])
        results.append(points)
    return results


def generate_duration_samples(team_samples, points_samples, duration_prob_lookup):
    results = []
    for team_sample, points_sample in zip(team_samples, points_samples):
        lookup = duration_prob_lookup[team_sample][points_sample]
        prob = norm(loc=lookup['mean'], scale=lookup['std'])
        sample = prob.rvs()

        # Don't allow negative durations
        while sample <= 0:
            sample = prob.rvs()
        results.append(sample)
    return results

team_samples = generate_team_samples(teams)
points_samples = generate_points_samples(team_points, team_samples)
duration_samples = generate_duration_samples(team_samples, points_samples, duration_
↪lookup)
```

We'll then save the data to a CSV using pandas so we can use it later if we need to.

```
import pandas as pd

# Convert the samples to a numpy array
data = np.array(list(zip(team_samples, points_samples, duration_samples)))

#write the numpy array to a csv using pandas
dataframe = pd.DataFrame(data=data, columns=['team', 'points', 'duration'])
dataframe.to_csv('duration_samples.csv')
```

Now we'll train our model. For this we'll use the GaussianProcessRegressorModel which wraps scikit-learn's GaussianProcessregressor.

```
from projectpredict.learningmodels import GaussianProcessRegressorModel
from projectpredict import TimeUnits
```

```

# By default, the kernel used in the model is
# ConstantKernel() + Matern(length_scale=1, nu=3 / 2) + WhiteKernel(noise_level=1)
# A custom jkernel can be specified using the "kernel" keyword in the constructor
model = GaussianProcessRegressorModel(TimeUnits.days)
input_data = data[data.columns.drop('duration')]
output = data['duration']

# Because we are using a pandas DataFrame, we don't need to specify the
# ordering of the data.
model.train(input_data, output)

# If we were using a raw numpy array or a python, we'd write
# model.train(input_data, output, ordering=['team', 'points'])
```

Now that model has been trained, we can add team and points data to our Tasks. Data is attached to Tasks using the "data" keyword argument in the constructor. The keys of the dictionary must be the same as the column names of the input data used to train the model, or the elements passed to the "ordering" keyword used to train the model.

```
from datetime import datetime

from projectpredict import Project, Task, TimeUnits, DatePdf
from projectpredict.pdf import GaussianPdf, DeterministicPdf


taskB_earliest_start_date = datetime(year=2018, month=5, day=14, hour=12)

taskA = Task('A', data={'team': 1, 'points': 3})

taskB = Task('B', data={'team': 3, 'points': 2},
    earliest_start_date_pdf=DatePdf(
        taskB_earliest_start_date,
        GaussianPdf(0, 2),
        units=TimeUnits.hours)
)

taskC = Task('C', data={'team': 2,'points': 1})

taskD_latest_finish_date = datetime(year=2018, month=5, day=16)
taskE_latest_finish_date = datetime(year=2018, month=5, day=15)
taskF_latest_finish_date = datetime(year=2018, month=5, day=20)


taskD = Task('D', data={'team': 4,'points': 3},
    latest_finish_date_pdf=DatePdf(
        taskD_latest_finish_date,
        GaussianPdf(0, 1),
        units=TimeUnits.days
    )
)

taskE = Task('E', data={'team': 1,'points': 2},
    latest_finish_date_pdf=DatePdf(taskE_latest_finish_date, DeterministicPdf(0))
)

taskF = Task('F', data={'team': 2,'points': 2},
    latest_finish_date_pdf=DatePdf(taskF_latest_finish_date, DeterministicPdf(0))
)
```

At this point, the tasks don't contain any estimates of their durations. We could set their duration estimates directly from the model using

```
taskA.set_duration_pdf(model)
```

But the add_task() and add_tasks() methods in the Project will automatically set the duration when it adds the Task(s) to the project, so we can use the same syntax as before with one slight modification: The project needs to be given the model in its constructor.

```
project = Project('MyProject', model=model)

tasks = [taskA, taskB, taskC, taskD, taskE, taskF]
dependencies = [
    (taskA, taskC),
    (taskB, taskC),
    (taskB, taskD),
    (taskC, taskE),
    (taskC, taskF)
]
project.add_tasks(tasks)
project.add_dependencies(dependencies)
```

We can then get the earliest finish date, latest start date, and total float in the same way as before.

```
current_time = datetime(year=2018, month=5, day=12, hour=12)

stats = project.calculate_task_statistics(current_time=current_time)
```

4.4 Updating Project Status

Now suppose the project begins, and we start with task A. We can mark it as started by doing the following

```
taskA_start_time = datetime(year=2018, month=5, day=13)

# Without specifying a start_time, the current wall time will be used
taskA.start(start_time=taskA_start_time)
```

Let's suppose that the task is completed 12 hours later, then we can mark it as complete by writing the following:

```
from datetime import timedelta
current_time = taskA_start_time + timedelta(hours=12)
taskA.complete(completion_time=current_time)
```

Marking a task as completed effectively removes it from the sampling and calculations of the earliest finish date, latest start date, and total float.

4.5 Recommendations

Now that we have completed Task A, the question then becomes what is the next Task which should be attempted. We can get recommendations from the project using the Project's recommend_next() method. For more information on the algorithm see *The Recommendation Engine*

```
project.recommend_next(current_time=current_time)
>>> (<Task name=B>,)
```

We can also get a recommendation for multiple tasks using the "max_number" keyword (there is also a corresponding "min_number" keyword).

```
project.recommend_next(current_time=current_time, max_number=2)
```

By default this batch mode recommendation system assumes that if a task in this batch is completed, a new task can begin immediately. To disable this behavior, set the "batch_wait" keyword to True.

```
project.recommend_next(current_time=current_time, max_number=2, batch_wait=True)
```

4.5.1 Customizing the Default Recommendation Algorithm

The default recommendation engine can be modified by setting a "risk_tolerance" score. This is a value between 0 and 1. The higher the score, the more emphasis is put on reducing the total float and less emphasis is put on the precision of the total float. The default is 0.5, but you can select your own by adding the "risk_tolerance" entry to the "selection_func_arguments" keyword argument.

```
project.recommend_next(
   current_time=current_time,
   max_number=2,
   selection_func_arguments={'risk_tolerance': 0.75}
)
```

You can also place more emphasis on certain deadlines than others, so if one task is critical to meet a deadline, you can specify a "deadline_weight" for a task by adding the keyword argument to the Task constructor. For example, to place more weight on meeting Task E's deadline, we could construct it as

```
taskE = Task('E', data={'team': 1,'points': 2},
    latest_finish_date_pdf=DatePdf(taskE_latest_finish_date, DeterministicPdf(0)),
    deadline_weight=10
)
```

4.5.2 Recommendations with Constraints

You can also limit the set of accepted tasks by adding constraint functions. Suppose you know that your velocity for a a sprint is 7 points. To restrict the set of tasks to ones wose story point sum is less than or equal to 7, you can construct a constraint function like the following

```
def story_point_constraint(project, task_set):
   story_point_sum = sum(task.data['points'] for task in task_set)
   return story_point_sum <= 7

project.recommend_next(
   current_time=current_time,
   max_number=2,
   constraints=[story_point_constraint]
)
```

4.5.3 Recommendations with Custom Scoring

You can also specify a custom scoring mechanism by specifying two function - a scoring function and a selection function. The scoring function must accept a dict in which the keys is a Task and the value is a list of TaskSamples generated by the sampling algorithm. Additional arguments can be accepted as keyword arguments to the recommend_next() method and will be forwarded to the scoring function. The recommendation selection function must accept a dict in which the keys are a tuple of Tasks and the value is the returned score from the scoring function. Additional arguments can be specified by supplying a dict of the arguments to the "selection_func_arguments" keyword argument of the recommend_next() method.

```
def my_score_func(samples, **score_args):
    foo = score_args['foo']
    bar = score_args['bar']
    # ...
    return some_score

def my_selection_func(scores, **selection_args):
    wiz = selection_args['wiz']
    bang = selection_args['bang']
    # ...
    return best_task

project.recommend_next(
    current_time=current_time,
    max_number=2,
    score_func=my_score_func,
    selection_func=my_selection_func,
    selection_func_arguments={'wiz': 0.75, 'bang': 'wizbang'}
    foo=12,
    bar='high_risk'
)
```

CHAPTER 5

Visualization

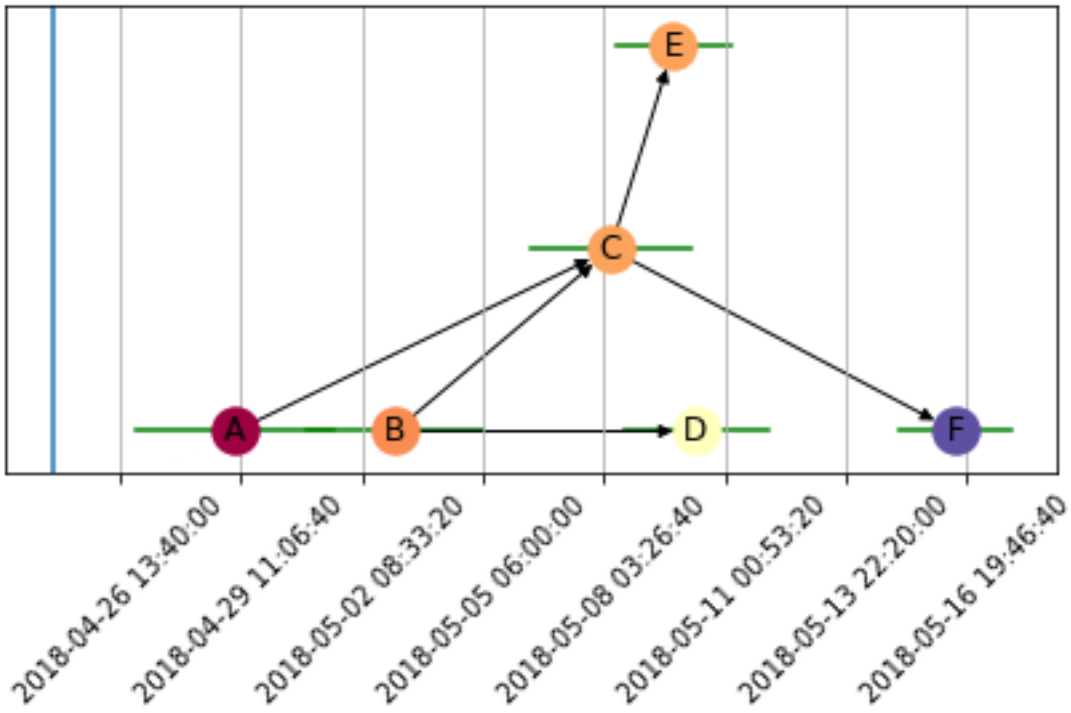
Currently only one artist, the MatplotlibArtist, is provided by ProjectPredict. It provides a single visualization of a project based on its generated statistics using matplotlib. It places positions the tasks on a graph based on its mean latest start date, creating a timeline of the project. Additionally, it can shade the tasks based on either the total float, latest start, or earliest finish (the default colormap is Matplotlib's Spectral colormap).

Note: The following example uses the Project developed using the learning model from *Learned Model*

```
from projectpredict.artists import MatplotlibArtist
import matplotlib.pyplot as plt

artist = MatplotlibArtist(project)
current_time = datetime(year=2018, month=4, day=25)
fig, ax = artist.draw(current_time=current_time)
plt.tight_layout()
plt.savefig('myproject.png')
```

This results in the following plot:



The horizontal bars indicate the standard deviation of the latest start date, and teh blue vertical bar represents the current date. These can be toggled off by setting the "show_variance=False" and "show_current_time=False" keyword arguments respectively.

5.1 Custom Visualizations

No interface must be satisfied to make your own visualizations, but an ArtistBase class has been provided which supplies a function, get_positions(), which generates a timeline-like graph of the project based on the latest start date for each task in the project. You can choose to extend from this base class or not.

5.2 Layout Algorithm

Constructing the visual layout of the Project is non trivial, and the current implementation still doesn't get it quite right. Currently the algorithm iterates through the tasks in topological order,

```
find the optimal spacing for the tasks
initialize the position of the first task (in topological order) to be 0,0
for task in topological sort of project:
   x_position = task's latest start date
   relevant_positions = all previously-seen tasks such that their x-distance is <=␣
↪the optimal distance
   if any of relevant_positions are predecessors of the current task:
      relevant_positions = the predecessors of the task which are in relevant_
↪positions
   best_neighbor = the task in relevant_positions whose x-position difference from␣
↪the current task is greatest
   y_position = y such that (x-position, y) is on a circle centered at best_neighbor␣
↪with radius optimal_distance
```

```
   store x_position, y_position for the task
```

The optimal distance is rather arbitrarily found by

```
start_tasks = all tasks with no predecessors
terminal_tasks = all tasks with no successors
max_path = longest path between any start task and any terminal task
max_time_difference = (end of max_path's latest finish date - start of max path's␣
↪latest finish date)
optimal_distance = max_time_difference / length of max_path
```

CHAPTER 6

Customized PDFs

ProjectPredict only comes with two built in PDFs, the DeterministicPdf and the GaussianPdf, however, making a custom PDF is straightforward, and requires only a minimal interface.

6.1 PDFs from Scipy

Generating custom PDFs from scipy.stats distributions requires only that you extend from the projectpredict.pdf.SciPyPdf base class and provide a constructor. For example, to provide a half-normal distribution from scipy.stats.halfnorm, you could write the following class

```
from scipy.stats import halfnorm
from math import sqrt

class HalfNormalPdf(SciPyPdf):
    def __init__(mean, variance):
        super(HalfNormalPdf, self).__init__(halfnorm(loc=mean, scale=sqrt(variance)))
```

6.2 Fully Custom PDFs

All PDFs must provide the following methods:

- A method called sample() which takes no parameters and return a random sample from the PDF in the form of a float
- A field or property called “mean” which holds the mean of the pdf
- A field or property called “variance” which holds the variance of the pdf

For example, a uniform PDF from Python’s built-in random module could be written as

```
from rand import uniform

class UniformPdf(object):
    def __init__(low, high):
        self.low = low
        self.high = high
        self.mean = (high - low) / 2
        self.variance = 1/12 * (high - low)**2

    def sample();
      return uniform(low, high)
```

CHAPTER 7

Customized Learning Models

ProjectPredict comes with a Gaussian Process Regression model, however you may find this model unsuitable for your data. To make your own model, you only need to follow a minimal interface – the only requirement is that you have a method named "predict" that accepts the dictionary of data associated with a task and returns a DurationPdf. For simplicity, assume your tasks have a "points" value in their data, and your model simply returns a DurationPdf wrapping a DeterministicPdf containing with the same value as the points passed into it. You could write this as

```
class SimpleModel(object):
    def __init__(self, units=TimeUnits.hours):
        self.units = units

    def predict(self, input_data):
        return DurationPdf(DeterministicPdf(input_data['points']), units=self.units)
```

CHAPTER 8

Next Steps

ProjectPredict is still in development, and numerous improvement can be made. Amoung them are:

- The default learning algorithm, the Gaussian Process Regressor model from scikit-learn does not perform adequately for a wide variety of data sets. Some alternatives would be to use GPFlow or pymc3 to determine the distribution using non-parametric Bayesian methods.
- The visualization capabilities are admittedly somewhat primitive and lacks the ability to interact with the project graph. A much better solution would be to set up a small web server and use cytoscape to view and interact with the model.
- Durations are internally represented as Python datetime.timedelta objects. It might be better to allow users to specify how long a working day is (8 hours) and define a day to be the length of the working hours.
- Completing a task should update the model so that it learns as the project progresses.

CHAPTER 9

Sphinx AutoAPI Index

This page is the top-level of your generated API documentation. Below is a list of all items that are documented here.

9.1 artists

9.1.1 Module Contents

class artists.**ArtistBase**(*project*)

Base class for artists. Contains methods to help determine the positions of the tasks

project

Project – The project to draw

Parameters **project** (Project) – The project to draw

__init__(*project*)

_date_to_timestamp()

_find_optimal_distance(*stats*)

Finds the best distance between nodes.

This is determined from the number of tasks in the longest path between all starting tasks and all terminal tasks. The optimal distance is the difference between the earliest latest finish date mean and the latest latest finish date mean divided by the number of nodes in the path.

Parameters **(dict{Task**(*stats*) – TaskStatistics}): The statistics used to derive the optimal distance

Returns The optimal distance between nodes.

Return type float

_find_longest_path_length(*start_tasks*, *terminal_tasks*)

get_positions(*stats*)

_find_best_y_position(*optimal_distance*, *positions*, *task*, *x_position*)
The optimal Y position is found by first finding the best task for the new task to be positioned near and solving the equation for a circle centered at that task's position with a radius equal to the optimal_distance for the y-variable.

_calculate_y_position(*x_position*, *optimal_distance*)

_get_relevant_positions(*task*, *positions*, *x_position*, *optimal_distance*)

_find_best_neighbor_task()

class artists.**MatplotlibArtist**(*project*)
Draws a project using Matplotlib

Note: There are still several issues with this artist. The task labels only fit a single letter, so the names often overflow. And the labels are too long and are improperly oriented.

---

project
Project – The project to draw

Parameters **project** (Project) – The project to draw

__init__(*project*)

_get_color_converter(*bounds*, *low_better*, *colormap*)

draw(*shade="total_float"*, *stats=None*, *current_time=None*, *iterations=1000*, *colormap="Spectral"*, *show_plot=True*, *show_variance=True*, *show_current_time=True*)
Draws a project and shades it by derived stats.

The X position of the tasks is determined by their latest start date

Parameters

- **shade** (*str*) – Shades the nodes by a derived stat. Accepted values are 'total_float', 'latest_start', or 'earliest_finish'
- **stats** (*list[TaskStatistics], optional*) – The statistics used to draw the Project. If none are supplied, the Project will be sampled.
- **current_time** (*datetime, optional*) – The current time to sample the Project. Only used if stats is not specified. Defaults to the current (UTC) time.
- **iterations** (*int, optional*) – The number of iterations to sample the Project from. Only used if stats is not specified. Defaults to 1000
- **colormap** (*str, optional*) – The matplotlib color map to use. Defaults to 'Spectral'
- **show_plot** (*bool, optional*) – Show the plot? Defaults to True.
- **show_variance** (*bool, optional*) – Show the variance of the latest start date? Defaults to True.
- **show_current_time** (*bool, optional*) – Show the current time as a vertical line? Defaults to True.

Returns The figure and axis of the plot

Return type tuple

_adjust_ticks()

_create_color_converter(*colormap*, *shade*, *stats*)

_add_variance_bars(*positions*, *stats*)

9.2 pdf

9.2.1 Module Contents

class pdf.**SciPyPdf**(*pdf*)

__init__(*pdf*)

sample()
Get a sample from the PDF

Returns A sample from the PDF

Return type float

mean()
float: The mean of the PDF

variance()
float: The variance of the PDF

__eq__(*other*)

__repr__()

class pdf.**GaussianPdf**(*mean*, *variance*)
A PDF representing a Gaussian distribution

pdf
norm – The Gaussian pdf object

Parameters **pdf** (*norm*) – The Gaussian pdf object

__init__(*mean*, *variance*)

from_dict(*dict_in*)
Creates a GaussianPdf from a dictionary

Parameters **dict_in** (*dict*) – The dict to create the PDF from. Must contain keys for 'mean' and 'variance'

Returns The constructed Gaussian PDF

Return type *GaussianPdf*

to_dict()
Gets a dictionary representation of this PDF

Returns The dictionary representation of this PDF

Return type dict

class pdf.**DeterministicPdf**(*value*)
A PDF representing a Gaussian distribution

pdf
float – The exact value to be returned by the sample() function

Parameters **value** (*float*) – The exact value to be returned by the sample() function

__init__(*value*)

sample()
Get a sample from the PDF. Will always return the value passed into the constructor.

Returns The value passed into the constructor

Return type float

mean()
float: The mean of the PDF. Always equal to the value passed into the constructor

variance()
float: The variance of the PDF. Will always return 0

__eq__(*other*)

from_dict(*dict_in*)
Creates a DeterministicPdf from a dictionary

Parameters **dict_in** (*dict*) – The dict to create the PDF from. Must contain keys for 'mean'

Returns The constructed deterministic PDF

Return type *DeterministicPdf*

to_dict()
Gets a dictionary representation of this PDF

Returns The dictionary representation of this PDF

Return type dict

class pdf.**PdfFactory**
Factory to construct PDFs from dictionaries

create(*pdf_type*, *parameters*)
Create a PDF

Parameters

- **pdf_type** (*str*) – The type of PDF to construct. Must match an entry in the pdf_registry
- **parameters** (*dict*) – The parameters from which to construct the PDF from.

Returns The constructed PDF

class pdf.**TimeUnits**
Enum representing possible units of time

to_timedelta(*value*)
Converts a TimeUnits and a value to a timedelta

Parameters

- **units** (TimeUnits) – The units to use with the timedelta
- **value** (*float*) – The value to use in the timedelta

Returns The timedelta with the given units and value

Return type timedelta

from_string(*value*)
Converts a string to a TimeUnits

Parameters **value** (*str*) – The string to convert

Returns The converted timeunit

Return type *TimeUnits*

Raises ValueError – If no matching string is found.

class pdf.**DurationPdf**(*pdf*, *units=None*)

A probability density function over a time duration

pdf

A probability density function object which provides a mechanism for sampling via a sample() method

units

TimeUnits – The units to use for the duration

Parameters

- **pdf** – A probability density function object
- **units** (TimeUnits, *optional*) – The units to use for the duration. Defaults to TimeUnits.seconds

__init__(*pdf*, *units=None*)

mean()

timedelta: The mean value of this PDF

sample(*minimum=None*)

Get a sample from the distribution

Parameters **minimum** (*timedelta*) – The minimum duration

Returns A sample from the distribution

Return type timedelta

__eq__(*other*)

class pdf.**DatePdf**(*mean_datetime*, *pdf*, *units=None*)

A probability density function over a datetime.

mean_datetime

datetime – A datetime to use as the mean value

pdf

A probability density function object which provides a sampling mechanism via a sample() method

units

TimeUnits – The units to use for the pdf samples

Parameters

- **mean_datetime** (*datetime*) – A datetime to use as the mean value
- **pdf** – A probability density function object
- **units** (TimeUnits, *optional*) – The units to use for pdf samples. Defaults to TimeUnits.seconds

__init__(*mean_datetime*, *pdf*, *units=None*)

mean()

timedelta: The mean value of this PDF

sample()
Get a sample from the distribution

Returns A sample from the distribution

Return type datetime

__eq__(*other*)

9.3 task

9.3.1 Module Contents

class task.**Entity**(*uid=None*, *name=""*)
Base class for entities which provides a UUID and a hashability to the child classes

uid
UUID – The UUID of the object

name
str – The name of the object

Parameters

- **uid** (*UUID, optional*) – The UUID of the object
- **name** (*str, optional*) – The name of the object

__init__(*uid=None*, *name=""*)

__eq__(*other*)

__hash__()

__repr__()

class task.**Task**(*name*, *uid=None*, *project_uid=None*, *duration_pdf=None*, *earliest_start_date_pdf=None*, *latest_finish_date_pdf=None*, *data=None*, *deadline_weight=1*)
A task in the project or overall process

project_uid
UUID – The UUID of the project containing this task

duration_pdf
projectpredict.pdf.DurationPdf – A pdf to use to sample the duration of the task

earliest_start_date_pdf
projectpredict.pdf.DatePdf – A pdf to use to sample the earliest start date of of the task

latest_finish_date_pdf
projectpredict.pdf.DatePdf – A pdf to use to sample the latest finish date of the task

start_time
datetime – The datetime the task was started

completion_time
datetime – The datetime the task was completed

data
Any data associated with this task.

deadline_weight
The weight attached to the deadline for this task.

Parameters

- **name** (*str*) – The name of the task
- **uid** (*UUID, optional*) – The UUID of the task. If none is provided, one will be generated.
- **project_uid** (*UUID, optional*) – The UUID of the project containing this task
- **duration_pdf** (*projectpredict.pdf.DurationPdf*) – A pdf to use to sample the duration of the task
- **earliest_start_date_pdf** (DatePdf, *optional*) – A pdf to use to sample the earliest start date of of the task
- **latest_finish_date_pdf** (DatePdf, *optional*) – A pdf to use to sample the latest finish date of the task
- **data** (*optional*) – Any data associated with this task.
- **deadline_weight** (*int, optional*) – The weight attached to meeting this task's deadline

__init__(*name*, *uid=None*, *project_uid=None*, *duration_pdf=None*, *earliest_start_date_pdf=None*, *latest_finish_date_pdf=None*, *data=None*, *deadline_weight=1*)

start(*start_time=None*)
Marks the task as started

Parameters **start_time** (*datetime, optional*) – The datetime the task was started. Defaults to the current UTC timestamp

complete(*completion_time=None*)
Completes the task

Parameters **completion_time** (*datetime, optional*) – The datetime the task was completed. Defaults to the current UTC timestamp

is_completed()
bool: Is the task completed?

is_started()
bool: Has the task been started?

mean_duration()
timedelta: Gets the mean of the duration pdf

set_duration_pdf(*model*)
Sets the duration PDF from a model

Parameters **model** – The model to use to predict the duration of the task

set_earliest_start_pdf(*mean_datetime*, *std*, *units=None*)
Sets the earliest start date pdf as a normal distributirequired=Trueon about a mean date.

Parameters

- **mean_datetime** (*datetime*) – The mean datetime of the earliest time a task can start
- **std** (*float*) – The standard deviation of the distribution

- **units** (*TimeUnits, optional*) – The units of time of the variance. Defaults to TimeUnits.seconds

set_latest_finish_pdf(*mean_datetime*, *std*, *units=None*)
Sets the latest finish date pdf as a normal distribution about a mean date.

Parameters

- **mean_datetime** (*datetime*) – The mean datetime of the latest time a task can finish
- **std** (*float*) – The standard deviation of the distribution
- **units** (*TimeUnits, optional*) – The units of time of the variance. Defaults to TimeUnits.seconds

get_duration_sample(*current_time*)
Gets a sample of the duration.

If the task has already started, then only durations greater than current_time - start_time will be valid, and samples will be drawn until a valid duration is picked.

Parameters **current_time** (*datetime*) – The current time at which the sample should be drawn from.

Returns A sample of the duration pdf

Return type timedelta

get_earliest_start_sample(*current_time*)
Gets a sample of the earliest start date pdf

If a task has been started, this will always return the start time. Else if an earliest start date pdf has been provided, a sample is drawn from that distribution. If no distribution has ben provided, the current time is returned.

Parameters **current_time** (*datetime*) – The current time at which the sample should be drawn from.

Returns A sample from the earliest start date pdf.

Return type datetime

get_latest_finish_sample()
Gets a sample of the latest finish date pdf

If an latest finish date pdf has been provided, a sample is drawn from that distribution. else, this function will return None

Returns A sample from the latest start date pdf

Return type datetime

from_pert(*name*, *best_case*, *estimated*, *worst_case*, *units=None*, ***kwargs*)
Constructs a Task from three-point (PERT) estimations.

Parameters

- **name** (*str*) – The name of the task
- **best_case** (*float*) – The estimated best case duration of the task
- **estimated** (*float*) – The estimated duration of the task
- **worst_case** (*float*) – The estimated worst case duration of the task
- **units** (*TimeUnits, optional*) – The units of time used in the estimation. Defaults to TimeUnits.seconds

- ****kwargs** – Arguments to be passed into Task constructor

Returns A task constructed from the provided arguments

Return type *Task*

9.4 exceptions

9.4.1 Module Contents

class InvalidProject(*errors*)
Exception thrown when a project is determined to be invalid

errors
list[str]|str – The errors found with the Project

Parameters **errors** (*list[str]|str*) – The errors found with the Project

__init__(*errors*)

__repr__()

9.5 project

9.5.1 Module Contents

project.**datetime_stats**(*datetimes*)
Gets the mean and variance of a collection of datetimes

Parameters **datetimes** (*iterable(datetime)*) – The datetimes to compute the statistics on.

Returns

A dictionary containing keys for the mean and variance. The mean is a datetime, and the variance is a timedelta.

Return type dict

project.**timedelta_stats**(*timedeltas*)
Gets the mean and variance of a collection of timedeltas

Parameters **timedeltas** (*iterable(timedelta)*) – The timedeltas to compute the statistics on.

Returns A dictionary containing keys for the mean and variance. both the mean and variance are datetimes.

Return type dict

class project.**TaskSample**(*duration*, *earliest_start*, *latest_finish*)
A wrapper for a sample of the derived statistics for a Task

duration
timedelta – The sampled duration of the task

earliest_start
datetime – The sampled earliest start date of the task

latest_finish
datetime – The sampled latest finish date of the task

latest_start
datetime – The latest start date of the task. Must be set independently of the constructor

earliest_finish
datetime – The earliest finish date of the task. Must be set independently of the constructor.

Parameters

- **duration** (*timedelta*) – The sampled duration of the task
- **earliest_start** (*datetime*) – The sampled earliest start date of the task
- **latest_finish** (*datetime*) – The sampled latest finish date of the task

__init__(*duration*, *earliest_start*, *latest_finish*)

total_float()
timedelta: The total float of the task. Earliest finish mst be set before calculation.

from_task(*task*, *current_time*)
Constructs a TaskSample from a task

Parameters

- **task** (Task) – The task to sample
- **current_time** (*datetime*) – The current datetime used to sample the task

Returns The constructed sample

Return type *TaskSample*

class project.**TaskStatistics**(*latest_start*, *earliest_finish*, *total_float*)
A container for the relevant derived statistics for a Task

latest_start
dict – A dict containing the mean and variance of the latest start date of the task in 'mean' and 'variance' keys respectively.

earliest_finish
dict – A dict containing the mean and variance of the earliest finish date of the task in 'mean' and 'variance' keys respectively.

total_float
dict – A dict containing the mean and variance of the total float date of the task in 'mean' and 'variance' keys respectively.

Parameters

- **latest_start** (*dict*) – A dict containing the mean and variance of the latest start date of the task in 'mean' and 'variance' keys respectively.
- **earliest_finish** (*dict*) – A dict containing the mean and variance of the earliest finish date of the task in 'mean' and 'variance' keys respectively.
- **total_float** (*dict*) – A dict containing the mean and variance of the total float date of the task in 'mean' and 'variance' keys respectively.

__init__(*latest_start*, *earliest_finish*, *total_float*)

from_samples(*samples*)
Construct a TaskStatistics object from samples

Parameters **samples** (*iterable(TaskSample)*) – The samples to compute the statistics from.

Returns The constructed TaskStatistics

Return type *TaskStatistics*

__repr__()

class project.**Project**(*name*, *model=None*, *uid=None*, *tasks=None*, *dependencies=None*)
A project

Note: This must be an acyclic graph.

---

name
str – The name of the project

uid
UUID – The UUID of the project

model
A model used to predict the duration of tasks from their data

Parameters

- **name** (*str*) – The name of the project
- **model** (*optional*) – A model used to predict the duration of tasks from their data
- **uid** (*UUID, optional*) – The UUID of the project
- **tasks** (*iterable(Task), optional*) – A collections of Tasks associated with this project
- **dependencies** (*iterable(dict), optional*) – The dependencies associated with the project in the form of dicts of 'source' and 'destination' keys.

__init__(*name*, *model=None*, *uid=None*, *tasks=None*, *dependencies=None*)

validate()
Validates the Project meets the requirements to do inference

Checks: * The Project is a directed acyclic graph * Every terminal Task (one without successors) has a latest start date PDF

Raises *InvalidProject* – If the project does not conform to the requirements.

dependencies()
list[tuple(Task, Task)]: The dependencies in the project where the first element of the tuple is the source task and the second element of the tuple is the dependent task.

tasks()
iterable(Task): The tasks of this project

dependencies_summary()
list[DependencySummary]: The dependencies of this project

get_task_from_id(*id_*)
Gets a task from an id

Parameters **id** (*UUID*) – The UUID of the project to get

Returns The task with the associated with the id or None if task is not found

Return type Task|None

add_task(*task*)
Adds a Task to this Project and determines the duration PDF of the task from the model if not previously specified.

Parameters **task** (Task) – The Task to add to the project

add_tasks(*tasks*)

Adds multiple Tasks to this Project and determines the duration PDF of the task from the model if not previously specified.

Parameters **tasks** (*iterable(Task)*) – The Task to add to the project

add_dependency(*parent*, *child*)
Adds a Task dependency to this Project

Parameters

- **parent** (Task) – The parent task
- **child** (Task) – The child task, i.e. the Task which depends on the parent

add_dependencies(*dependencies*)
Adds multiple Task dependencies to this Project

Parameters **dependencies** (*list[tuple(Task, Task)]*) – A list of tuples of Task dependencies in the form of (parent task, child task)

calculate_earliest_finish_times(*current_time=None*, *iterations=1000*)
Generates samples of the earliest finish times for each uncompleted node in the project.

Parameters

- **current_time** (*datetime*) – the time at which to take the samples
- **iterations** (*int, optional*) – The number of samples to generate. Defaults to 1000

Returns [datetime]}: A dictionary of the samples for each task.

Return type dict{Task

earliest_finish_sample_func(*parents*, *children*, *samples*, ***kwargs*)

calculate_latest_start_times(*iterations=1000*)
Generates samples of the latest start times for each uncompleted node in the project.

Parameters **iterations** (*int, optional*) – The number of samples to generate. Defaults to 1000

Returns [datetime]}: A dictionary of the samples for each task.

Return type dict{Task

latest_start_sample_func(*parents*, *children*, *samples*, ***kwargs*)

_get_samples(*forward_sample_func=None*, *backward_sample_func=None*, *iterations=1000*, *current_time=None*, ***kwargs*)

_get_parents_and_children(*task*)

calculate_task_statistics(*current_time=None*, *iterations=1000*)

recommend_next(*current_time=None*, *constraints=None*, *iterations=1000*, *score_func=None*, *selection_func=None*, *min_number=1*, *max_number=1*, *batch_wait=False*, *selection_func_arguments=None*, ***score_func_arguments*)

Get the recommended next tasks

Parameters

- **current_time** (*datetime, optional*) – The current time (in UTC) to query the project. Defaults to the current time.
- **constraints** (*iterable(callable)*) – A list of constraints to apply to the selected tasks. These must be functions which task in two parameters – the project (self) and the set of Tasks under consideration.
- **iterations** (*int, optional*) – The number of iterations to query the project for each considered set of Tasks. Defaults to 1000.
- **score_func** (*func, optional*) – The function used to score the results of a Task set. Defaults to a function which returns a dict containing the mean and precision (inverse variance) of the total float of each task weighted by the Tasks' deadline weight. The function must take keyword arguments which can be specified as keyword arguments to this function (see score_func_arguments).
- **selection_func** (*func, optional*) – The function used to select which task set is best from the results returned from the score_func. Defaults to a function which scales the total float and precision each between 0 and 1 and sums them according to a weighting parameter (see selection_func_arguments). The function must accept a dict of Task set to score and keyword arguments which can be specified by the selection_func_arguments parameter of this function.
- **min_number** (*int, optional*) – The minimum number of tasks which can can be recommended. Defaults to 1.
- **max_number** (*int, optional*) – The maximum number of tasks which can be recommended. Defaults to 1.
- **batch_wait** (*bool, optional*) – Do all tasks for a proposed tuple of Tasks need to be completed before the next tasks can begin? Defaults to False.
- **selection_func_arguments** (*dict, optional*) – The arguments to be passed to the selection_func.
- ****score_func_arguments*** – The arguments to pass to the score_func

Returns The recommended tasks to complete next

Return type tuple(*Task*)

recommendation_sample_func(*parents*, *children*, *samples*, ***kwargs*)

_default_recommendation_score_func(***kwargs*)

_default_recommendation_selection_func(***kwargs*)

get_starting_and_terminal_tasks()

Gets the starting tasks (ones without predecessors) and terminal tasks (ones without successors)

Returns

The starting and terminal tasks in the form of (starting tasks, terminal tasks)

Return type tuple(list[*Task*], list[*Task*])

update_from_dict(*data*)
Updates the Project using a dictionary of new values

Parameters **data** (*dict*) – The new values

from_dict(*data_in*, *model*)
Constructs a Project from a dictionary of values and a model

Parameters

- **data_in** (*dict*) – The data to construct the Project from
- **model** – The model used to predict the durations of tasks

Returns The constructed project

Return type *Project*

9.6 learningmodels

9.6.1 Submodules

learningmodels.scikit

Module Contents

class learningmodels.scikit.**GaussianProcessRegressorModel**(*units=None*, ***kwargs*)
Learns the duration of a task from data using scikit-learn's GaussianProcessRegressor

model
GaussianProcessRegressor – The underlying model used to predict the data

units
TimeUnits, optional – The time units the resulting durations should be in. Defaults to TimeUnits.seconds

is_trained
bool – A boolean value indicating if the model has been trained.

ordering
list[str] – The ordering of the input data used to construct input data

Parameters **units** (TimeUnits, *optional*) – The time units the resulting durations should be in. Defaults to TimeUnits.seconds

Keyword Arguments **kernel** – The kernel to use in the regressor model. Defaults to ConstantKernel() + Matern(length_scale=1, nu=3 / 2) + WhiteKernel(noise_level=1)

__init__(*units=None*, ***kwargs*)

train(*input_data*, *durations*, *ordering=None*)
Trains the model from input data and durations

Note: If a Pandas DataFrame is used for the input data, the ordering of the data will be determined by the ordering of the colunms. If a pandas DataFrame is not used, then the ordering will need to be

provided. Each Task must provide data as a dictionary in which the keys are the same as the names in the ordering/column names of the DataFrame

Parameters

- **input_data** (*array-like*) – The data to train the data from
- **durations** (*array-like*) – The durations associated with the data
- **ordering** (*list[str], optional*) – The ordering of the data

Raises `ValueError` – When a non-DataFrame is provided as the input_data and no ordering is provided

predict(*input_data*)

Predicts the duration of a task given its data

Parameters

- **input_data** (*dict*) – A dict containing the data necessary to predict the duration. The format must be as
- **pairs in which the key is the name of the data and the value is its value.** (*key-value*) –

Returns The estimated duration of the task.

Return type *DurationPdf*

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

Symbols

A

C

D

U

V