Example to plot directly into latex

19-10-2019

1 Introduction

Welcome, this document presents our market analysis for the TruCol consultancy. The objective of this document is to provide some basic insight into the order of magnitude of the potential of the TruCol consultancy to generate returns for its potential investors. Based on various pitch templates, [2], and private communications, we intend to convey this information through sharing our model and estimate of the following market parameters for the TruCol consultancy:

- **Total addressable market (TAM)**, or total available market, is the total market demand for a product or service, calculated in annual revenue or unit sales if 100% of the available market is achieved[1].
- **Serviceable available market (SAM)** is the portion of TAM targeted and served by a company's products or services[1].
- **Serviceable obtainable market (SOM)**, or share of market, is the percentage of SAM which is realistically reached[1].

Since we currently have little experience on this topic within our team we are making our data and assumptions as transparant as possible, both in this document as in our code. This way we hope to improve our model based on your feedback by enabling you tingle with it yourself. Additionally, because the market analysis consists of a rough estimate, three different estimation methods are used for generating the TAM, SAM and SOM estimates. The redundancy is introduced to establish some referenceframe within the results.

The assumptions and datapoints for the respective models are specified in section 2. Next, the models are described in section 3 (the Python models themselves are included as appendices in section D to section F respectively). The results of these models are presented in section 4. To shed some light on how sensitive the model is to for example changes in assumptions, a sensitivity analysis is presented for each model in section 5. Next the results and sensitivity of the models are discussed in section 6 and a conclusion is provided in section 7.

We invite you to tinker with the assumptions and models yourself! The data and plots in this report are automatically updated if you run `python -m code.project1.src`. If you experience any difficulties in running the code, simply reach out to us, (click on issues on the github page) and we are happy to get you running the code.

2 Assumptions

2.1 Top Down

2.2 Bottom Up

2.3 Value Theory

3 Model Description

3.1 Market

To compute the TAM, SAM and SOM, some form of market definition can be used. To this end it can be valuable to specify exactly what the TruCol consultancy does, where it adds value and how it does that. Furthermore, since these three afore mentioned estimates pertain to a potential future, the potential, yet deemed feasible, activities of the TruCol consultancy are included.

The TruCol consultancy provides advice and support to companies on how companies can get the most out of the TruCol protocol. To understand this the following assumptions are shared:

- **asu-0:** Task completions of tasks that are completed using the TruCol protocol are deterministically verifyable.

- **asu-1:** Solutions of tasks that are completed using the TruCol protocol are of sufficient quality.
- **asu-2:** Tasks that are completed using the TruCol protocol can be solved for the lowest costprice that is currently available in this world.
- **asu-3:** No personel needs to be attracted, screened, hired nor fired for tasks that are completed using the TruCol protocol.
- **asu-4:** Companies can benefit from public particular solutions to their task specifications.
- **asu-5:** By sampling from a bigger talent pool (this world), the average performance of the solutions will be better than what is produced by the in-house talent pool, or, for equal solution performance, a faster rate of development can be obtained on average for an equal or lower price.

Based on these assumptions, one can conclude that an economically rational company would try to off-load as much of their required tasks into the TruCol protocol as it would minimise their operational costs.

We help companies identify the tassks for which they can use the TruCol protocol, and we assist them in writing safe test specifications that are not easily hackable.

3.2 Market Size

3.3 Market Trajectory

Since the market size estimation models are somewhat of an abstract/subjective task, three different approaches are used in an attempt to establish some reference material with respect to accuracy.

Before the model is presented, it is important to realise that we propose a consultancy service that operates as an optimisation service. This means that if a certain activity, e.g. a logistics company has operational cost of 5 $million/day, our consultancy service is only able to earn at most the margin of improvement we are able to bring our customer. So suppose the independent usage of the TruCol provides the customer with a 2% optimisation in their operational costs, yielding them $5.000.000 \cdot 0.02 = 100.000/day\$$. Suppose our expertise is able to enable them to yield a 3% optimisation by identifying the relevant development/system processes and supporting them in improved test specification. In that assumption our consultancy would bring them an additional 3-2=1% which would translate roughly to 50.000$. That would be the value we bring to the logistics company in this hypothetical scenario.

In reality this example is oversimplified, the 2% the company could get by themselves would involve some risk pertaining to inaccurate test specification which could lead to loss of the bounty. Our company reduces this risk by providing test-specification security expertise. Furthermore, our interaction with the client may bring the client experience that can be applied in future applications of the TruCol protocol, hence the value to we bring to the client is larger than the amount they gain in terms of optimisation w.r.t. the case where they use the protocol themselves.

3.3.1 Top Down

The Top-Down approach

3.3.2 Bottom Up

3.3.3 Value Theory

;

4 Results

4.1 Top Down

4.2 Top Down

4.3 Top Down

5 Sensitivity Analysis

5.1 Top Down

5.2 Bottom Up

5.3 Value Theory

6 Discussion

6.1 Top Down

6.2 Bottom Up

6.3 Value Theory

7 Conclusion

References

[1] The businessplan shop. Tam sam som - what it means and why it matters.

[2] Haje Jan Kamps. *Pitch Deck Design*. Apress, Berkeley, CA, 2020.

A Appendix __main__.py

```python
import os
from .Main import Main

print(f"Hi, I'll be running the main code, and I'll let you know when I'm done.")
project_nr = 1
main = Main()

# export the code to latex
main.export_code_to_latex(project_nr)

# compile the latex report
main.compile_latex_report(project_nr)

print(f"Done.")
```

B Appendix Main.py

```python
# Example code that creates plots directly in report
# Code is an implementation of a genetic algorithm
import random
from matplotlib import pyplot as plt
from matplotlib import lines
import matplotlib.pyplot as plt
import numpy as np

from .Compile_latex import Compile_latex
from .Plot_to_tex import Plot_to_tex as plt_tex
from .Export_code_to_latex import export_code_to_latex

# define global variables for genetic algorithm example
string_length = 100
mutation_chance = 1.0 / string_length
max_iterations = 1500


class Main:
    def __init__(self):
        pass

    def export_code_to_latex(self, project_nr):
        export_code_to_latex("main.tex", project_nr)

    def compile_latex_report(self, project_nr):
        """compiles latex code to pdf"""
        compile_latex = Compile_latex(project_nr, "main.tex")

    def addTwo(self, x):
        """adds two to the incoming integer and returns the result of
           ↪ the computation."""
        return x + 2


if __name__ == "__main__":
    # initialize main class
    main = Main()
```

C Appendix Compile_latex.py

```python
# runs a jupyter notebook and converts it to pdf

import os
import shutil
import nbformat
from nbconvert.preprocessors import ExecutePreprocessor


class Compile_latex:
    def __init__(self, project_nr, latex_filename):
        self.script_dir = self.get_script_dir()
        relative_dir = f"latex/project{project_nr}/"
        self.compile_latex(relative_dir, latex_filename)
        self.clean_up_after_compilation(latex_filename)
        self.move_pdf_into_latex_dir(relative_dir, latex_filename)

    # runs jupyter notebook
    def compile_latex(self, relative_dir, latex_filename):
        os.system(f"pdflatex {relative_dir}{latex_filename}")

    def clean_up_after_compilation(self, latex_filename):
        latex_filename_without_extention = latex_filename[:-4]
        print(f"latex_filename_without_extention={
            ↪ latex_filename_without_extention}")
        self.delete_file_if_exists(f"{
            ↪ latex_filename_without_extention}.aux")
        self.delete_file_if_exists(f"{
            ↪ latex_filename_without_extention}.log")
        self.delete_file_if_exists(f"texput.log")

    def move_pdf_into_latex_dir(self, relative_dir, latex_filename):
        pdf_filename = f"{latex_filename[:-4]}.pdf"
        destination = f"{self.get_script_dir()}/../../../{
            ↪ relative_dir}{pdf_filename}"

        try:
            shutil.move(pdf_filename, destination)
        except:
            print("Error while moving file ", pdf_filename)

    def delete_file_if_exists(self, filename):
        try:
            os.remove(filename)
        except:
            print(
                f"Error while deleting file: {filename} but that is
                    ↪ not too bad because the intention is for it to
                    ↪ not be there."
            )

    def get_script_dir(self):
        """returns the directory of this script regardles of from
            ↪ which level the code is executed"""
        return os.path.dirname(__file__)


if __name__ == "__main__":
    main = Compile_latex()
```

D Appendix Export_code_to_latex.py

```python
# runs a jupyter notebook and converts it to pdf
import os
import shutil
import nbformat
from nbconvert.preprocessors import ExecutePreprocessor


def export_code_to_latex(main_latex_filename, project_nr):
    """This function exports the python files and compiled pdfs of jupiter notebooks into the
    latex of the same project number. First it scans which appendices (without code, without
    notebooks) are already manually included in the main latex code. Next, all appendices
    that contain the python code are eiter found or created in the following order:
    First, the __main__.py file is included, followed by the main.py file, followed by all
    python code files in alphabetic order. After this, all the pdfs of the compiled notebooks
    are added in alphabetic order of filename. This order of appendices is overwritten in the
    main tex file.

    :param main_latex_filename: Name of the main latex document of this project number
    :param project_nr: The number  indicating which project this code pertains to.
    """
    script_dir = get_script_dir()
    relative_dir = f"latex/project{project_nr}/"
    appendix_dir = script_dir + "/../../../" + relative_dir + "Appendices/"
    path_to_main_latex_file = (
        f"{script_dir}/../../../{relative_dir}/{main_latex_filename}"
    )
    root_dir = script_dir[0 : script_dir.rfind(f"code/project{project_nr}")]

    # Get paths to files containing python code.
    python_filepaths = get_filenames_in_dir("py", script_dir, ["__init__.py"])
    compiled_notebook_pdf_filepaths = get_compiled_notebook_paths(script_dir)

    # Check which files are already included in the latex appendicess.
    python_files_already_included_in_appendices = get_code_files_already_included_in_appendices(
        python_filepaths, appendix_dir, ".py", project_nr, root_dir
    )
    notebook_pdf_files_already_included_in_appendices = get_code_files_already_included_in_appendices(
        compiled_notebook_pdf_filepaths, appendix_dir, ".ipynb", project_nr, root_dir,
    )

    # Get which appendices are still missing.
    missing_python_files_in_appendices = get_code_files_not_yet_included_in_appendices(
```

```
        python_filepaths, python_files_already_included_in_appendices, ".py"
    )
    missing_notebook_files_in_appendices = get_code_files_not_yet_included_in_appendices(
        compiled_notebook_pdf_filepaths,
        notebook_pdf_files_already_included_in_appendices,
        ".pdf",
    )

    # Create the missing appendices.
    created_python_appendix_filenames = create_appendices_with_code(
        appendix_dir, missing_python_files_in_appendices, ".py", project_nr, root_dir
    )

    created_notebook_appendix_filenames = create_appendices_with_code(
        appendix_dir,
        missing_notebook_files_in_appendices,
        ".ipynb",
        project_nr,
        root_dir,
    )

    appendices = get_list_of_appendix_files(
        appendix_dir, compiled_notebook_pdf_filepaths, python_filepaths
    )

    main_tex_code, start_index, end_index, appendix_tex_code = get_appendix_tex_code(
        path_to_main_latex_file
    )

    # assumes non-included non-code appendices should not be included:
    # overwrite the existing appendix lists with the current appendix list.
    (
        non_code_appendices,
        main_non_code_appendix_inclusion_lines,
    ) = get_order_of_non_code_appendices_in_main(appendices, appendix_tex_code)

    python_appendix_filenames = list(
        map(
            lambda x: x.appendix_filename,
            filter_appendices_by_type(appendices, "python"),
        )
    )
    sorted_created_python_appendices = sort_python_appendices(
        filter_appendices_by_type(appendices, "python")
    )
    sorted_python_appendix_filenames = list(
        map(lambda x: x.appendix_filename, sorted_created_python_appendices)
    )

    notebook_appendix_filenames = list(
        map(
            lambda x: x.appendix_filename,
```

```
95                 filter_appendices_by_type(appendices, "notebook"),
96             )
97         )
98         sorted_created_notebook_appendices =
               ↪ sort_notebook_appendices_alphabetically(
99             filter_appendices_by_type(appendices, "notebook")
100        )
101        sorted_notebook_appendix_filenames = list(
102            map(lambda x: x.appendix_filename,
                   ↪ sorted_created_notebook_appendices)
103        )
104
105        appendix_latex_code = create_appendices_latex_code(
106            main_non_code_appendix_inclusion_lines,
107            sorted_created_notebook_appendices,
108            project_nr,
109            sorted_created_python_appendices,
110        )
111
112        updated_main_tex_code = substitute_appendix_code(
113            end_index, main_tex_code, start_index, appendix_latex_code
114        )
115        print(f'\n\n')
116        print(f"updated_main_tex_code={updated_main_tex_code}")
117
118        overwrite_content_to_file(updated_main_tex_code,
               ↪ path_to_main_latex_file)
119
120
121    def create_appendices_latex_code(
122        main_non_code_appendix_inclusion_lines,
123        notebook_appendices,
124        project_nr,
125        python_appendices,
126    ):
127        """Creates the latex code that includeds the appendices in the
               ↪ main latex file.
128
129        :param main_non_code_appendix_inclusion_lines: latex code that
               ↪ includes the appendices that do not contain python code nor
               ↪  notebooks
130        :param notebook_appendices: List of Appendix objects representing
               ↪  appendices that include the pdf files of compiled Jupiter
               ↪ notebooks
131        :param project_nr: The number indicating which project this code
               ↪ pertains to.
132        :param python_appendices: List of Appendix objects representing
               ↪ appendices that include the python code files.
133        """
134        main_appendix_inclusion_lines =
               ↪ main_non_code_appendix_inclusion_lines
135        print(f"main_appendix_inclusion_lines={
               ↪ main_appendix_inclusion_lines}")
136
137        appendices_of_all_types = [python_appendices, notebook_appendices
               ↪ ]
138
139        print(f"\n\n")
140        main_appendix_inclusion_lines.append(
141            f"\IfFileExists{{latex/project{project_nr}/main.tex}}{{"
142        )
143        main_appendix_inclusion_lines = append_latex_inclusion_command(
```

```
144         appendices_of_all_types, True, main_appendix_inclusion_lines,
                ↪  project_nr,
145     )
146     main_appendix_inclusion_lines.append(f"}}{{")
147     main_appendix_inclusion_lines = append_latex_inclusion_command(
148         appendices_of_all_types, False, main_appendix_inclusion_lines
                ↪ , project_nr,
149     )
150     #main_appendix_inclusion_lines.append(f"}}")
151     print(f"main_appendix_inclusion_lines={
            ↪ main_appendix_inclusion_lines}")
152     return main_appendix_inclusion_lines
153
154
155 def append_latex_inclusion_command(
156     appendices_of_all_types, is_from_root_dir,
            ↪ main_appendix_inclusion_lines, project_nr
157 ):
158     for appendix_type in appendices_of_all_types:
159         for appendix in appendix_type:
160             line = update_appendix_tex_code(
161                 appendix.appendix_filename, is_from_root_dir,
                        ↪ project_nr
162             )
163             print(f"appendix.appendix_filename={appendix.
                    ↪ appendix_filename}")
164             main_appendix_inclusion_lines.append(line)
165     return main_appendix_inclusion_lines
166
167
168 def filter_appendices_by_type(appendices, appendix_type):
169     """Returns the list of all appendices of a certain appendix type,
            ↪  from the incoming list of Appendix objects.
170
171     :param appendices: List of Appendix objects
172     :param appendix_type: Can consist of "no_code", "python", or "
            ↪ notebook" and indicates different appendix types
173     """
174     return_appendices = []
175     for appendix in appendices:
176         if appendix.appendix_type == appendix_type:
177             return_appendices.append(appendix)
178     return return_appendices
179
180
181 def sort_python_appendices(appendices):
182     """First puts __main__.py, followed by main.py followed by a-z
            ↪ code files.
183
184     :param appendices: List of Appendix objects
185     """
186     return_appendices = []
187     for appendix in appendices:  # first get appendix containing
            ↪ __main__.py
188         if (appendix.code_filename == "__main__.py") or (
189             appendix.code_filename == "__Main__.py"
190         ):
191             return_appendices.append(appendix)
192             appendices.remove(appendix)
193     for appendix in appendices:  # second get appendix containing
            ↪ main.py
194         if (appendix.code_filename == "main.py") or (
```

```
            appendix.code_filename == "Main.py"
        ):
            return_appendices.append(appendix)
            appendices.remove(appendix)
    return_appendices

    # Filter remaining appendices in order of a-z
    filtered_remaining_appendices = [
        i for i in appendices if i.code_filename is not None
    ]
    appendices_sorted_a_z = sort_appendices_on_code_filename(
        filtered_remaining_appendices
    )
    return return_appendices + appendices_sorted_a_z


def sort_notebook_appendices_alphabetically(appendices):
    """Sorts notebook appendix objects alphabetic order of their pdf
       ↪ filenames.

    :param appendices: List of Appendix objects
    """
    return_appendices = []
    filtered_remaining_appendices = [
        i for i in appendices if i.code_filename is not None
    ]
    appendices_sorted_a_z = sort_appendices_on_code_filename(
        filtered_remaining_appendices
    )
    return return_appendices + appendices_sorted_a_z


def sort_appendices_on_code_filename(appendices):
    """Returns a list of Appendix objects that are sorted and  based
       ↪ on the property: code_filename.
    Assumes the incoming appendices only contain python files.

    :param appendices: List of Appendix objects
    """
    attributes = list(map(lambda x: x.code_filename, appendices))
    sorted_indices = sorted(range(len(attributes)), key=lambda k:
       ↪ attributes[k])
    sorted_list = []
    for i in sorted_indices:
        sorted_list.append(appendices[i])
    return sorted_list


def get_order_of_non_code_appendices_in_main(appendices,
   ↪ appendix_tex_code):
    """Scans the lines of appendices in the main code, and returns
       ↪ the lines
    of the appendices that do not contain code, in the order in which
       ↪  they were
    included in the main latex file.

    :param appendices: List of Appendix objects
    :param appendix_tex_code: latex code from the main latex file
       ↪ that includes the appendices
    """
    non_code_appendices = []
    non_code_appendix_lines = []
```

```
250     appendix_tex_code = list(dict.fromkeys(appendix_tex_code))
251     for line in appendix_tex_code:
252         appendix_filename = get_filename_from_latex_appendix_line(
            ↪ appendices, line)
253 
254         # Check if line is not commented
255         if not appendix_filename is None:
256             if not line_is_commented(line, appendix_filename):
257                 appendix = get_appendix_from_filename(appendices,
                    ↪ appendix_filename)
258                 if appendix.appendix_type == "no_code":
259                     non_code_appendices.append(appendix)
260                     non_code_appendix_lines.append(line)
261     return non_code_appendices, non_code_appendix_lines
262 
263 
264 def get_filename_from_latex_appendix_line(appendices, appendix_line):
265     """Returns the first filename from a list of incoming filenames
        ↪ that
266     occurs in a latex code line.
267 
268     :param appendices: List of Appendix objects
269     :param appendix_line: latex code (in particular expected to be
        ↪ the code from main that is used to include appendix latex
        ↪ files.)
270     """
271     for filename in list(map(lambda appendix: appendix.
        ↪ appendix_filename, appendices)):
272         if filename in appendix_line:
273             if not line_is_commented(appendix_line, filename):
274                 return filename
275 
276 
277 def get_appendix_from_filename(appendices, appendix_filename):
278     """Returns the first Appendix object with an appendix filename
        ↪ that matches the incoming appendix_filename.
279     The Appendix objects are selected from an incoming list of
        ↪ Appendix objects.
280 
281     :param appendices: List of Appendix objects
282     :param appendix_filename: name of a latex appendix file, ends in
        ↪ .tex,
283     """
284     for appendix in appendices:
285         if appendix_filename == appendix.appendix_filename:
286             return appendix
287 
288 
289 def get_compiled_notebook_paths(script_dir):
290     """Returns the list of jupiter notebook filepaths that were
        ↪ compiled successfully and that are
291     included in the same dias this script (the src directory).
292 
293     :param script_dir: absolute path of this file.
294     """
295     notebook_filepaths = get_filenames_in_dir(".ipynb", script_dir)
296     compiled_notebook_filepaths = []
297 
298     # check if the jupyter notebooks were compiled
299     for notebook_filepath in notebook_filepaths:
300 
301         # swap file extension
```

```
302             notebook_filepath = notebook_filepath.replace(".ipynb", ".pdf
                    ↪ ")
303 
304             # check if file exists
305             if os.path.isfile(notebook_filepath):
306                 compiled_notebook_filepaths.append(notebook_filepath)
307         return compiled_notebook_filepaths
308 
309 
310 def get_list_of_appendix_files(
311         appendix_dir, absolute_notebook_filepaths,
                ↪ absolute_python_filepaths
312 ):
313         """Returns a list of Appendix objects that contain all the
                ↪ appendix files with .tex extension.
314 
315         :param appendix_dir: Absolute path that contains the appendix .
                ↪ tex files.
316         :param absolute_notebook_filepaths: List of absolute paths to the
                ↪  compiled notebook pdf files.
317         :param absolute_python_filepaths: List of absolute paths to the
                ↪ python files.
318         """
319         appendices = []
320         appendices_paths = get_filenames_in_dir(".tex", appendix_dir)
321 
322         for appendix_filepath in appendices_paths:
323             appendix_type = "no_code"
324             appendix_filecontent = read_file(appendix_filepath)
325             line_nr_python_file_inclusion = get_line_of_latex_command(
326                 appendix_filecontent, "\pythonexternal{"
327             )
328             line_nr_notebook_file_inclusion = get_line_of_latex_command(
329                 appendix_filecontent, "\includepdf[pages="
330             )
331             if line_nr_python_file_inclusion > -1:
332                 appendix_type = "python"
333                 # get python filename
334                 line = appendix_filecontent[line_nr_python_file_inclusion
                        ↪ ]
335                 filename = get_filename_from_latex_inclusion_command(
336                     line, ".py", "\pythonexternal{"
337                 )
338                 appendices.append(
339                     Appendix(
340                         appendix_filepath,
341                         appendix_filecontent,
342                         appendix_type,
343                         filename,
344                         line,
345                     )
346                 )
347             if line_nr_notebook_file_inclusion > -1:
348                 appendix_type = "notebook"
349                 line = appendix_filecontent[
                        ↪ line_nr_notebook_file_inclusion]
350                 filename = get_filename_from_latex_inclusion_command(
351                     line, ".pdf", "\includepdf[pages="
352                 )
353                 appendices.append(
354                     Appendix(
355                         appendix_filepath,
```

```
                    appendix_filecontent,
                    appendix_type,
                    filename,
                    line,
                )
            )
        else:
            appendices.append(
                Appendix(appendix_filepath, appendix_filecontent,
                    ↪ appendix_type)
            )
    return appendices


def get_filename_from_latex_inclusion_command(
    appendix_line, extension, start_substring
):
    """returns the code/notebook filename in a latex command which
        ↪ includes that code in an appendix.
    The inclusion command includes a python code or jupiter notebook
        ↪ pdf.

    :param appendix_line: :Line of latex code (in particular expected
        ↪  to be the latex code from an appendix.).
    :param extension: The file extension of the file that is sought
        ↪ in the appendix line. Either ".py" or ".pdf".
    :param start_substring: The substring that characterises the
        ↪ latex inclusion command.
    """
    start_index = appendix_line.index(start_substring)
    end_index = appendix_line.index(extension)
    return get_filename_from_dir(
        appendix_line[start_index : end_index + len(extension)]
    )


def get_filenames_in_dir(extension, path, excluded_files=None):
    """Returns a list of the relative paths to all files within the
        ↪ some path that match
    the given file extension.

    :param extension: The file extension of the file that is sought
        ↪ in the appendix line. Either ".py" or ".pdf".
    :param path: Absolute filepath in which files are being sought.
    :param excluded_files: (Default value = None) Files that will not
        ↪  be included even if they are found.
    """
    filepaths = []
    for r, d, f in os.walk(path):
        for file in f:
            if file.endswith(extension):
                if (excluded_files is None) or (
                    (not excluded_files is None) and (not file in
                        ↪ excluded_files)
                ):
                    filepaths.append(r + "/" + file)
    return filepaths


def get_code_files_already_included_in_appendices(
    absolute_code_filepaths, appendix_dir, extension, project_nr,
        ↪ root_dir
```

```
):
    """Returns a list of code filepaths that are already properly
       ↪ included the latex appendix files of this project.

    :param absolute_code_filepaths: List of absolute paths to the
       ↪ code files (either python files or compiled jupyter
       ↪ notebook pdfs).
    :param appendix_dir: Absolute path that contains the appendix .
       ↪ tex files.
    :param extension: The file extension of the file that is sought
       ↪ in the appendix line. Either ".py" or ".pdf".
    :param project_nr: The number  indicating which project this code
       ↪  pertains to.
    :param root_dir: The root directory of this repository.
    """
    appendix_files = get_filenames_in_dir(".tex", appendix_dir)
    contained_codes = []
    for code_filepath in absolute_code_filepaths:
        for appendix_filepath in appendix_files:
            appendix_filecontent = read_file(appendix_filepath)
            line_nr = check_if_appendix_contains_file(
                appendix_filecontent, code_filepath, extension,
                   ↪ project_nr, root_dir
            )
            if line_nr > -1:
                # add filepath to list of files that are already in
                   ↪ the appendices
                contained_codes.append(
                    Appendix_with_code(
                        code_filepath,
                        appendix_filepath,
                        appendix_filecontent,
                        line_nr,
                        ".py",
                    )
                )
    return contained_codes


def check_if_appendix_contains_file(
    appendix_content, code_filepath, extension, project_nr, root_dir
):
    """Scans an appendix content to determine whether it contains a
       ↪ substring that
    includes a code file (of either python or compiled notebook=pdf
       ↪ extension).

    :param appendix_content: content in an appendix latex file.
    :param code_filepath: Absolute path to a code file (either python
       ↪  files or compiled jupyter notebook pdfs).
    :param extension: The file extension of the file that is sought
       ↪ in the appendix line. Either ".py" or ".pdf".
    :param project_nr: The number  indicating which project this code
       ↪  pertains to.
    :param root_dir: The root directory of this repository.
    """
    # convert code_filepath to the inclusion format in latex format
    latex_relative_filepath = (
        f"latex/project{project_nr}/../../{code_filepath[len(root_dir
           ↪ ):]}"
    )
```

```
454     latex_command = get_latex_inclusion_command(extension,
          ↪ latex_relative_filepath)
455     return get_line_of_latex_command(appendix_content, latex_command)
456
457
458 def get_line_of_latex_command(appendix_content, latex_command):
459     """Returns the line number of a latex command if it is found.
          ↪ Returns -1 otherwise.
460
461     :param appendix_content: content in an appendix latex file.
462     :param latex_command: A line of latex code. (Expected to come
          ↪ from some appendix)
463     """
464     # check if the file is in the latex code
465     line_nr = 0
466     for line in appendix_content:
467         if latex_command in line:
468             if line_is_commented(line, latex_command):
469                 commented = True
470             else:
471                 return line_nr
472         line_nr = line_nr + 1
473     return -1
474
475
476 def line_is_commented(line, target_substring):
477     """Returns True if a latex code line is commented, returns False
          ↪ otherwise
478
479     :param line: A line of latex code that contains a relevant
          ↪ command (target substring).
480     :param target_substring: Used to determine whether the command
          ↪ that is found is commented or not.
481     """
482     left_of_command = line[: line.rfind(target_substring)]
483     if "%" in left_of_command:
484         return True
485     return False
486
487
488 def get_latex_inclusion_command(extension,
      ↪ latex_relative_filepath_to_codefile):
489     """Creates and returns a latex command that includes either a
          ↪ python file or a compiled jupiter
490     notebook pdf (whereever the command is placed). The command is
          ↪ intended to be placed in the appendix.
491
492     :param extension: The file extension of the file that is sought
          ↪ in the appendix line. Either ".py" or ".pdf".
493     :param latex_relative_filepath_to_codefile: The latex compilation
          ↪  requires a relative path towards code files
494     that are included. Therefore, a relative path towards the code is
          ↪  given.
495     """
496     if extension == ".py":
497         left = "\pythonexternal{"
498         right = "}"
499         latex_command = f"{left}{latex_relative_filepath_to_codefile
              ↪ }{right}"
500     elif extension == ".ipynb":
501
502         left = "\includepdf[pages=-]{"
```

```
503         right = "}"
504         latex_command = f"{left}{latex_relative_filepath_to_codefile
              ↪ }{right}"
505     return latex_command
506
507
508 def read_file(filepath):
509     """Reads content of a file and returns it as a list of strings,
          ↪ with one string per line.
510
511     :param filepath: path towards the file that is being read.
512     """
513     with open(filepath) as f:
514         content = f.readlines()
515     return content
516
517
518 def get_code_files_not_yet_included_in_appendices(
519     code_filepaths, contained_codes, extension
520 ):
521     """Returns a list of filepaths that are not yet properly included
          ↪  in some appendix of this project.
522
523     :param code_filepath: Absolute path to all the code files in
          ↪ this project (source directory).
524     (either python files or compiled jupyter notebook pdfs).
525     :param contained_codes: list of  Appendix objects that include
          ↪ either python files or compiled jupyter notebook pdfs,
          ↪ which
526     are already included in the appendix tex files. (Does not care
          ↪ whether those appendices are also actually
527     included in the main or not.)
528     :param extension: The file extension of the file that is sought
          ↪ in the appendix line. Either ".py" or ".pdf".
529     """
530     contained_filepaths = list(
531         map(lambda contained_file: contained_file.code_filepath,
              ↪ contained_codes)
532     )
533     not_contained = []
534     for filepath in code_filepaths:
535         if not filepath in contained_filepaths:
536             not_contained.append(filepath)
537     return not_contained
538
539
540 def create_appendices_with_code(
541     appendix_dir, code_filepaths, extension, project_nr, root_dir
542 ):
543     """Creates the latex appendix files in with relevant codes
          ↪ included.
544
545     :param appendix_dir: Absolute path that contains the appendix .
          ↪ tex files.
546     :param code_filepaths: Absolute path to code files that are not
          ↪ yet included in an appendix
547     (either python files or compiled jupyter notebook pdfs).
548     :param extension: The file extension of the file that is sought
          ↪ in the appendix line. Either ".py" or ".pdf".
549     :param project_nr: The number  indicating which project this code
          ↪  pertains to.
550     :param root_dir: The root directory of this repository.
```

```
    """
    appendix_filenames = []
    appendix_reference_index = (
        get_index_of_auto_generated_appendices(appendix_dir,
            ↪ extension) + 1
    )

    for code_filepath in code_filepaths:
        latex_relative_filepath = (
            f"latex/project{project_nr}/../../{code_filepath[len(
                ↪ root_dir):]}"
        )
        code_path_from_latex_main_path = f"../../{code_filepath[len(
            ↪ root_dir):]}"
        content = []
        filename = get_filename_from_dir(code_filepath)

        content = create_section(appendix_reference_index, filename,
            ↪ content)
        content = add_include_code_in_appendix(
            content,
            code_filepath,
            code_path_from_latex_main_path,
            extension,
            latex_relative_filepath,
            project_nr,
            root_dir,
        )

        print(f"content={content}")

        overwrite_content_to_file(
            content,
            f"{appendix_dir}Auto_generated_{extension[1:]}_App{
                ↪ appendix_reference_index}.tex",
            False,
        )
        appendix_filenames.append(
            f"Auto_generated_{extension[1:]}_App{
                ↪ appendix_reference_index}.tex"
        )
        appendix_reference_index = appendix_reference_index + 1
    return appendix_filenames


def add_include_code_in_appendix(
    content,
    code_filepath,
    code_path_from_latex_main_path,
    extension,
    latex_relative_filepath,
    project_nr,
    root_dir,
):
    """Includes the latex code that includes code in the script.

    :param content: The latex content that is being written to an
        ↪ appendix.
    :param code_path_from_latex_main_path: the path to the code as
        ↪ seen from the folder that contains main.tex.
    :param extension: The file extension of the file that is sought
        ↪ in the appendix line. Either ".py" or ".pdf".
```

```
    :param latex_relative_filepath_to_codefile: The latex compilation
       ↪  requires a relative path towards code files
    that are included. Therefore, a relative path towards the code is
       ↪  given.
    """
    print(f"before={content}")
    # TODO: append if exists}
    content.append(
        f"\IfFileExists{{latex/project{project_nr}/../../{
           ↪ code_filepath[len(root_dir):]}}}{{"
    )
    # append current line
    content.append(get_latex_inclusion_command(extension,
       ↪ latex_relative_filepath))
    # TODO: append {}
    content.append(f"}}{{")
    # TODO: code_path_from latex line
    content.append(
        get_latex_inclusion_command(extension,
           ↪ code_path_from_latex_main_path)
    )
    # TODO: add closing bracket }
    content.append(f"}}")
    print(f"after={content}")
    return content


def get_index_of_auto_generated_appendices(appendix_dir, extension):
    """Returns the maximum index of auto generated appendices of
    a specific extension type.

    :param extension: The file extension of the file that is sought
       ↪ in the appendix line. Either ".py" or ".pdf".
    :param appendix_dir: Absolute path that contains the appendix .
       ↪ tex files.
    """
    max_index = -1
    appendices =
       ↪ get_auto_generated_appendix_filenames_of_specific_extension
       ↪ (
        appendix_dir, extension
    )
    for appendix in appendices:
        substring = f"Auto_generated_{extension[1:]}_App"
        # remove left of index
        remainder = appendix[appendix.rfind(substring) + len(
           ↪ substring) :]
        # remove right of index
        index = int(remainder[:-4])
        if index > max_index:
            max_index = index
    return max_index


def get_auto_generated_appendix_filenames_of_specific_extension(
    appendix_dir, extension
):
    """Returns the list of auto generated appendices of
    a specific extension type.

    :param extension: The file extension of the file that is sought
       ↪ in the appendix line. Either ".py" or ".pdf".
```

```
655     :param appendix_dir: Absolute path that contains the appendix .
          ↪ tex files.
656     """
657     appendices_of_extension_type = []
658 
659     # get all appendices
660     appendix_files = get_filenames_in_dir(".tex", appendix_dir)
661 
662     # get appendices of particular extention type
663     for appendix_filepath in appendix_files:
664         right_of_slash = appendix_filepath[appendix_filepath.rfind("/
              ↪ ") + 1 :]
665         if (
666             right_of_slash[: 15 + len(extension) - 1]
667             == f"Auto_generated_{extension[1:]}"
668         ):
669             appendices_of_extension_type.append(appendix_filepath)
670     return appendices_of_extension_type
671 
672 
673 def create_section(appendix_reference_index, code_filename, content):
674     """Creates the header of a latex appendix file, such that it
          ↪ contains a section that
675     indicates the section is an appendix, and indicates which pyhon
          ↪ or notebook file is
676     being included in that appendix.
677 
678     :param appendix_reference_index: A counter that is used in the
          ↪ label to ensure the appendix section labels are unique.
679     :param code_filename: file name of the code file that is included
680     :param content: A list of strings that make up the appendix, with
          ↪  one line per element.
681     """
682     # write section
683     left = "\section{Appendix "
684     middle = code_filename.replace("_", "\_")
685     right = "}\label{app:"
686     end = "}"  # TODO: update appendix reference index
687     content.append(f"{left}{middle}{right}{appendix_reference_index}{
          ↪ end}")
688     return content
689 
690 
691 def overwrite_content_to_file(content, filepath, content_has_newlines
      ↪ =True):
692     """Writes a list of lines of tex code from the content argument
          ↪ to a .tex file
693     using overwriting method. The content has one line per element.
694 
695     :param content: The content that is being written to file.
696     :param filepath: Path towards the file that is being read.
697     :param content_has_newlines:  (Default value = True)
698     """
699     with open(filepath, "w") as f:
700         for line in content:
701             if content_has_newlines:
702                 f.write(line)
703             else:
704                 f.write(line + "\n")
705 
706 
707 def get_appendix_tex_code(main_latex_filename):
```

```
708     """gets the latex appendix code from the main tex file.
709 
710     :param main_latex_filename: Name of the main latex document of
        ↪ this project number
711     """
712     main_tex_code = read_file(main_latex_filename)
713     start = "\\begin{appendices}"
714     end = "\end{appendices}"
715     start_index = get_index_of_substring_in_list(main_tex_code, start
        ↪ ) + 1
716     end_index = get_index_of_substring_in_list(main_tex_code, end)
717     return main_tex_code, start_index, end_index, main_tex_code[
        ↪ start_index:end_index]
718 
719 
720 def get_index_of_substring_in_list(lines, target_substring):
721     """Returns the index of the line in which the first character of
        ↪ a latex substring if it is found
722     uncommented in the incoming list.
723 
724     :param lines: List of lines of latex code.
725     :param target_substring: Some latex command/code that is sought
        ↪ in the incoming text.
726     """
727     for i in range(0, len(lines)):
728         if target_substring in lines[i]:
729             if not line_is_commented(lines[i], target_substring):
730                 return i
731 
732 
733 def update_appendix_tex_code(appendix_filename, is_from_root_dir,
    ↪ project_nr):
734     """Returns the latex command that includes an appendix .tex file
        ↪ in an appendix environment
735     as can be used in the main tex file.
736 
737     :param appendix_filename: Name of the appendix that is included
        ↪ by the generated command.
738     :param project_nr: The number indicating which project this code
        ↪ pertains to.
739     """
740     if is_from_root_dir:
741         left = f"\input{{latex/project{project_nr}/"
742     else:
743         left = "\input{"
744     middle = "Appendices/"
745     right = "} \\newpage\n"
746     return f"{left}{middle}{appendix_filename}{right}"
747 
748 
749 def substitute_appendix_code(
750     end_index, main_tex_code, start_index,
        ↪ updated_appendices_tex_code
751 ):
752     """Replaces the old latex code that included the appendices in
        ↪ the main.tex file with the new latex
753     commands that include the appendices in the latex report.
754 
755     :param end_index: Index at which the appendix section ends right
        ↪ before the latex \end{appendix} line,
756     :param main_tex_code: The code that is saved in the main .tex
        ↪ file.
```

```
757     :param start_index: Index at which the appendix section starts
            ↪ right after the latex \begin{appendix} line,
758     :param updated_appendices_tex_code: The newly created code that
            ↪ includes all the relevant appendices.
759     (relevant being (in order): manually created appendices, python
            ↪ codes, pdfs of compiled jupiter notebooks).
760     """
761     updated_main_tex_code = (
762         main_tex_code[0:start_index]
763         + updated_appendices_tex_code
764         + main_tex_code[end_index:]
765     )
766     print(f"start_index={start_index}")
767     return updated_main_tex_code
768 
769 
770 def get_filename_from_dir(path):
771     """Returns a filename from an absolute path to a file.
772 
773     :param path: path to a file of which the name is queried.
774     """
775     return path[path.rfind("/") + 1 :]
776 
777 
778 def get_script_dir():
779     """returns the directory of this script regardles of from which
            ↪ level the code is executed"""
780     return os.path.dirname(__file__)
781 
782 
783 class Appendix_with_code:
784     """stores in which appendix file and accompanying line number in
            ↪ the appendix in which a code file is
785     already included. Does not take into account whether this
            ↪ appendix is in the main tex file or not
786     """
787 
788     def __init__(
789         self,
790         code_filepath,
791         appendix_filepath,
792         appendix_content,
793         file_line_nr,
794         extension,
795     ):
796         self.code_filepath = code_filepath
797         self.appendix_filepath = appendix_filepath
798         self.appendix_content = appendix_content
799         self.file_line_nr = file_line_nr
800         self.extension = extension
801 
802 
803 class Appendix:
804     """stores in appendix files and type of appendix."""
805 
806     def __init__(
807         self,
808         appendix_filepath,
809         appendix_content,
810         appendix_type,
811         code_filename=None,
812         appendix_inclusion_line=None,
```

```
813     ):
814         self.appendix_filepath = appendix_filepath
815         self.appendix_filename = get_filename_from_dir(self.
              ↪ appendix_filepath)
816         self.appendix_content = appendix_content
817         self.appendix_type = appendix_type  # TODO: perform
              ↪ validation of input values
818         self.code_filename = code_filename
819         self.appendix_inclusion_line = appendix_inclusion_line
```

E Appendix Model_bottom_up.py

```
# The bottom up model that computes the TAM and TSM
import random
from matplotlib import pyplot as plt
from matplotlib import lines
import matplotlib.pyplot as plt
import numpy as np

from .Plot_to_tex import Plot_to_tex as plt_tex


class Model_bottom_up:
    def __init__(self):
        pass

    def addTwo(self, x):
        """adds two to the incoming integer and returns the result of
           ↪ the computation."""
        return x + 2
```

F Appendix Model_top_down.py

```
# The bottom up model that computes the TAM and TSM
import random
from matplotlib import pyplot as plt
from matplotlib import lines
import matplotlib.pyplot as plt
import numpy as np

from .Plot_to_tex import Plot_to_tex as plt_tex


class Model_bottom_up:
    def __init__(self):
        pass

    def addTwo(self, x):
        """adds two to the incoming integer and returns the result of
           ↪ the computation."""
        return x + 2
```

G Appendix Model_value_theory.py

```python
# The bottom up model that computes the TAM and TSM
import random
from matplotlib import pyplot as plt
from matplotlib import lines
import matplotlib.pyplot as plt
import numpy as np

from .Plot_to_tex import Plot_to_tex as plt_tex


class Model_bottom_up:
    def __init__(self):
        pass

    def addTwo(self, x):
        """adds two to the incoming integer and returns the result of the computation."""
        return x + 2
```

H Appendix Plot_to_tex.py

```
### Call this from another file, for project 11, question 3b:
### from Plot_to_tex import Plot_to_tex as plt_tex
### multiple_y_series = np.zeros((nrOfDataSeries,nrOfDataPoints), dtype=int); # actually fill with data
### lineLabels = [] # add a label for each dataseries
### plt_tex.plotMultipleLines(plt_tex,single_x_series, multiple_y_series,"x-axis label [units]","y-axis label [units]",lineLabels,"3b",4,11)
### 4b=filename
### 4 = position of legend, e.g. top right.
###
### For a single line, use:
### plt_tex.plotSingleLine(plt_tex,range(0, len(dataseries)), dataseries,"x-axis label [units]","y-axis label [units]", lineLabel,"3b",4,11)

### You can also plot a table directly into latex, see example_create_a_table(..)
###
### Then put it in latex with for example:
###\begin{table}[H]
###     \centering
###     \caption{Results some computation.}\label{tab:some_computation}
###     \begin{tabular}{|c|c|} % remember to update this to show all columns of table
###         \hline
###         \input{latex/project3/tables/q2.txt}
###     \end{tabular}
###\end{table}
import random
from matplotlib import lines
import matplotlib.pyplot as plt
import numpy as np
import os


class Plot_to_tex:
    def __init__(self):
        self.script_dir = self.get_script_dir()
        print("Created main")

    # plot graph (legendPosition = integer 1 to 4)
    def plotSingleLine(
        self,
        x_path,
        y_series,
        x_axis_label,
        y_axis_label,
        label,
        filename,
        legendPosition,
        project_nr,
    ):
        fig = plt.figure()
        ax = fig.add_subplot(111)
        ax.plot(x_path, y_series, c="b", ls="-", label=label, fillstyle="none")
        plt.legend(loc=legendPosition)
        plt.xlabel(x_axis_label)
```

```
        plt.ylabel(y_axis_label)
        plt.savefig(
            os.path.dirname(__file__)
            + "/../../../latex/project"
            + str(project_nr)
            + "/Images/"
            + filename
            + ".png"
        )

    #         plt.show();

    # plot graphs
    def plotMultipleLines(
        self, x, y_series, x_label, y_label, label, filename,
            ↪ legendPosition, project_nr
    ):
        fig = plt.figure()
        ax = fig.add_subplot(111)

        # generate colours
        cmap = self.get_cmap(len(y_series[:, 0]))

        # generate line types
        lineTypes = self.generateLineTypes(y_series)

        for i in range(0, len(y_series)):
            # overwrite linetypes to single type
            lineTypes[i] = "-"
            ax.plot(
                x,
                y_series[i, :],
                ls=lineTypes[i],
                label=label[i],
                fillstyle="none",
                c=cmap(i),
            )
            # color

        # configure plot layout
        plt.legend(loc=legendPosition)
        plt.xlabel(x_label)
        plt.ylabel(y_label)
        plt.savefig(
            os.path.dirname(__file__)
            + "/../../../latex/project"
            + str(project_nr)
            + "/Images/"
            + filename
            + ".png"
        )

        print(f"plotted lines")

    # Generate random line colours
    # Source: https://stackoverflow.com/questions/14720331/how-to-
        ↪ generate-random-colors-in-matplotlib
    def get_cmap(n, name="hsv"):
        """Returns a function that maps each index in 0, 1, ..., n-1
            ↪ to a distinct
        RGB color; the keyword argument name must be a standard mpl
            ↪ colormap name."""
```

```
110         return plt.cm.get_cmap(name, n)
111
112     def generateLineTypes(y_series):
113         # generate varying linetypes
114         typeOfLines = list(lines.lineStyles.keys())
115
116         while len(y_series) > len(typeOfLines):
117             typeOfLines.append("-.")
118
119         # remove void lines
120         for i in range(0, len(y_series)):
121             if typeOfLines[i] == "None":
122                 typeOfLines[i] = "-"
123             if typeOfLines[i] == "":
124                 typeOfLines[i] = ":"
125             if typeOfLines[i] == " ":
126                 typeOfLines[i] = "--"
127         return typeOfLines
128
129     # Create a table with: table_matrix = np.zeros((4,4),dtype=object
        ↪ ) and pass it to this object
130     def put_table_in_tex(self, table_matrix, filename, project_nr):
131         cols = np.shape(table_matrix)[1]
132         format = "%s"
133         for col in range(1, cols):
134             format = format + " & %s"
135         format = format + ""
136         plt.savetxt(
137             os.path.dirname(__file__)
138             + "/../../../latex/project"
139             + str(project_nr)
140             + "/tables/"
141             + filename
142             + ".txt",
143             table_matrix,
144             delimiter=" & ",
145             fmt=format,
146             newline="  \\\\ \hline \n",
147         )
148
149     # replace this with your own table creation and then pass it to
        ↪ put_table_in_tex(..)
150     def example_create_a_table(self):
151         project_nr = "1"
152         table_name = "example_table_name"
153         rows = 2
154         columns = 4
155         table_matrix = np.zeros((rows, columns), dtype=object)
156         table_matrix[:, :] = ""  # replace the standard zeros with
            ↪ emtpy cell
157         print(table_matrix)
158         for column in range(0, columns):
159             for row in range(0, rows):
160                 table_matrix[row, column] = row + column
161         table_matrix[1, 0] = "example"
162         table_matrix[0, 1] = "grid sizes"
163
164         self.put_table_in_tex(table_matrix, table_name, project_nr)
165
166     def get_script_dir(self):
167         """returns the directory of this script regardles of from
            ↪ which level the code is executed"""
```

```
168         return os.path.dirname(__file__)
169
170
171 if __name__ == "__main__":
172     main = Plot_to_tex()
173     main.example_create_a_table()
```