8 Transforming data into decisions with linear programming

This chapter covers

- Linear programming
- Constrained optimization
- Objective functions
- Inequality and other constraints

In today's competitive business environment, managers and decision-makers are constantly challenged to make the most of limited resources. These resources can include time, money, labor, materials, and more. The goal is to maximize efficiency, profitability, or output, or, alternatively, to minimize costs, waste, or time. Achieving this requires careful planning and precise execution. One of the most effective tools for this purpose is linear programming.

Linear programming is a powerful mathematical approach to solving a specific type of constrained optimization problem, where both the objective and the constraints are represented by linear relationships. As a key tool in operations research and management science, linear programming has widespread applications across various industries. From optimizing supply chains to maximizing profits, linear

programming helps decision-makers find efficient solutions to complex, resource-limited challenges.

In essence, linear programming involves defining an objective function, which is a formula that needs to be maximized or minimized, subject to a set of constraints. These constraints represent the limitations or requirements that must be satisfied. By converting a real-world problem into a mathematical model, linear programming allows us to explore different scenarios and identify the optimal solution.

One of the most popular applications of linear programming is in the field of logistics and transportation. Companies use it to determine the most efficient way to distribute products from multiple warehouses to various retail locations, minimizing transportation costs while ensuring timely delivery. Another significant application is in manufacturing, where linear programming helps in scheduling production runs, balancing workloads, and minimizing waste.

More broadly, constrained optimization, a field that includes linear programming, focuses on finding the best solution within a set of given constraints. Constrained optimization is widely used across fields such as economics, engineering, finance, and even sports. For example, investors use constrained optimization to construct portfolios that maximize returns while adhering to risk tolerance levels and regulatory constraints. In engineering, constrained optimization helps design systems and components that meet performance standards while minimizing costs.

8.1 Problem formulation

In previous chapters, we solved real-world problems using real data. In this chapter, although the data will be illustrative, the challenge remains genuine and significant. Every organization faces the annual task of prioritizing which capital projects to pursue from a backlog of dozens, given limited budgets and resources. We will demonstrate the application of linear programming and constrained optimization techniques to strategically select projects that maximize value within given constraints.

To bring this scenario to life, we'll consider a fictional organization named Advanced Analytics and Infrastructure Management (AAIM), which is the data solutions provider for a sales and distribution company. AAIM specializes in reporting, advanced analytics, and the management of analytic infrastructure. This organization follows the Agile methodology to prioritize, plan, and execute its projects, referred to as features in Agile terminology.

Agile

In Agile methodology, work is structured in a hierarchical manner to ensure manageability and clarity. At the top of the hierarchy are *epics,* which are large bodies of work encompassing multiple *features.* An epic represents a high-level goal or objective that can span multiple sprints or even entire releases. To make these goals more

(continued)

achievable, epics are broken down into features. Features represent significant functionalities or pieces of work derived from epics. They are more detailed and can typically be completed within a release or across several sprints.

Each feature is further broken down into *user stories.* User stories are short, simple descriptions of functionalities from the perspective of the user, defining specific requirements. They are small enough to be completed within a single sprint. Finally, user stories are decomposed into *tasks,* which are the smallest unit of work in Agile. Tasks represent specific actions or steps required to complete a user story and are typically assigned to individual team members. This structured approach ensures that large objectives (epics) are systematically divided into manageable parts (features), detailed requirements (user stories), and actionable steps (tasks) for efficient execution and tracking.

8.1.1 The scenario

Every year, AAIM faces the challenge of selecting the most valuable features to work on from a comprehensive backlog. This selection process is crucial because resources such as budget, time, and workforce are always limited. Prioritizing the right features not only ensures efficient use of these resources but also aligns with the strategic objectives of the organization.

AAIM has shortlisted 20 features to deliberate over, each falling under one of three strategic objectives, or epics:

1. Enhance Data Infrastructure
2. Improve Data Analytics
3. Advance Machine Learning and AI Capabilities

To make informed decisions, AAIM will utilize linear programming techniques to balance its limited resources with its strategic goals, ensuring that each epic receives appropriate attention and resources. This structured approach operates under the assumption that linear constraints accurately represent resource limitations and relationships. Although linear programming is effective for many resource allocation problems, it's worth noting that in some cases, nonlinear constrained optimization may be more appropriate for capturing complex, nonlinear interactions. By using linear programming, however, AAIM can maximize its impact and drive forward its key objectives effectively.

8.1.2 The challenge

AAIM must prioritize and select features from those shortlisted from its backlog, ensuring that the selection process adheres to the following constraints:

- *Budget constraints*—The estimated cost of selected features must not exceed a specific limit for the year and overall.

- *Resource constraints*—The features selected must align with the available workforce and time.
- *Strategic balance*—Each epic should have a balanced representation of features to ensure all strategic objectives are advanced.

By adhering to these constraints, AAIM can strategically plan its project portfolio.

8.1.3 The approach

To address this challenge, AAIM will use linear programming to formulate and solve the problem. The steps involved in this process include the following:

- *Defining the objective function*—The objective function will aim to maximize the priority score of selected features, which is a measure of value derived from each feature relative to the effort required.
- *Setting the constraints*—Constraints will be defined for budget limits, resource availability, and strategic balance. Specifically, each epic should have a minimum of three and a maximum of four features selected.
- *Formulating the linear programming model*—By translating the real-world scenario into a precise mathematical model, AAIM can systematically represent its resources, constraints, and objectives. This modeling step is essential, as it allows AAIM to explore various scenarios and identify the optimal feature selection within its limitations, setting the stage for strategic and efficient decision-making.

This approach will ensure that AAIM effectively prioritizes and selects the most valuable projects to pursue, thereby making the best use of its limited resources while equally advancing its strategic objectives.

PRIORITY SCORES

One of our first tasks will be to calculate priority scores for selected features. To illustrate how priority scores are derived, let's consider one of AAIM's features: the Automated Data Cleaning Pipeline. This feature involves developing an automated data cleaning pipeline to preprocess raw data, handle missing values, and correct data inconsistencies. The prioritization process evaluates both the value and effort associated with this feature, resulting in a priority score that helps determine its value relative to other features.

The value assessment is as follows (each criterion is scored on a scale of 1 to 5, with 5 being the highest score):

1. Customer impact: high (5 points)
   - Automates a significant portion of the data preparation process, saving time for data scientists and improving productivity
2. Revenue generation: medium (3 points)
   - Enables faster data processing, leading to quicker insights and faster time-to-market for data-driven products

3. Strategic alignment: high (5 points)
   - Aligns with the organization's goal to improve operational efficiency and data quality
4. Risk mitigation: medium (3 points)
   - Reduces the risk of human error in data preprocessing and ensures more consistent data quality

Total value points: 16

The effort assessment is as follows (again, each criterion is scored on a scale of 1 to 5, with 5 being the highest score):

1. Complexity: high (5 points)
   - Requires significant development effort and testing to ensure the pipeline handles various data formats and edge cases
2. Dependencies: medium (3 points)
   - Depends on the availability of raw data sources and integration with existing data processing tools
3. Technical Challenge: high (5 points)
   - Involves advanced data engineering techniques and potential use of machine learning models for data imputation

Total effort points: 13

To prioritize features, AAIM calculates the priority score as the ratio of total value points to total effort points:

$$\text{Priority Score} = \frac{\text{Total Value Points}}{\text{Total Effort Points}}$$

Therefore, the priority score for the automated data cleaning pipeline feature equals

$$\text{Priority Score(Automated Data Cleaning Pipeline)} = \frac{16}{13} = 1.23$$

Despite the complexities and other challenges, the value proposition of the automated data cleaning pipeline feature still surpasses the estimated level of effort. However, its overall ranking among the other 19 features shortlisted from the backlog remains to be determined, particularly as we factor constraints into the decision-making process.

Agile and linear optimization

Agile methodology and linear optimization techniques, such as linear programming and constrained optimization, complement each other effectively in project management. Although not every Agile team constructs optimization frameworks, and not every organization using optimization techniques practices Agile, the integration of

these approaches can be highly beneficial. Agile employs quantitative methods to score features based on a combination of value and effort. These scores are crucial in prioritizing work and ensuring that the most impactful features are addressed first.

In Agile, features are often evaluated using metrics like customer impact, revenue generation, strategic alignment, and risk mitigation. Each feature receives a value score and an effort score. The value score represents the potential benefit, whereas the effort score indicates the resources required for implementation. This quantitative evaluation aligns perfectly with the principles of linear optimization.

The score derived from these Agile evaluations can serve as an objective function in a constrained optimization framework. By maximizing the priority score, which is the ratio of value to effort, organizations can ensure that their resources are allocated efficiently. Constraints such as budget limits, resource availability, and strategic balance across different epics can be incorporated into the optimization process.

Thus, Agile's quantitative scoring system provides a structured basis for developing a linear optimization framework, ensuring that project prioritization is both strategic and data-driven. This integration allows for intelligent selection of features, maximizing value while adhering to resource constraints.

This straightforward value-versus-effort calculation—not uncommon in Agile teams—offers a clear and structured approach for the team to make informed decisions about feature prioritization; in fact, it provides an objective function to use within our constrained optimization framework. By focusing on initiatives that offer the highest value relative to the effort invested, AAIM ensures that resources are allocated efficiently and effectively, maximizing their overall effect.

8.1.4 Feature summaries

Tables 8.1–8.3—one for each epic—list the features shortlisted from the backlog. Each table includes the estimated cost for the first year, the total cost to complete, and the priority score. The features are not listed in any particular order.

Table 8.1 Features aligned with the Enhance Data Infrastructure epic

Feature name	First-year cost	Total cost	Priority score
Automated Data Cleaning Pipeline	$50,000	$80,000	1.23
Real-time Data Ingestion	$70,000	$120,000	1.29
Data Privacy and Compliance Tools	$80,000	$130,000	1.19
Real-time Data Monitoring and Alerts	$60,000	$100,000	1.38
Integration with External Data Sources	$50,000	$85,000	1.42
Feature Engineering Automation	$55,000	$90,000	1.45

Table 8.2 Features aligned with the Improve Data Analytics and Reporting epic

Feature name	First-year cost	Total cost	Priority score
Interactive Data Visualization Dashboard	$60,000	$100,000	1.67
Predictive Maintenance Model	$55,000	$90,000	1.13
Fraud Detection System	$80,000	$130,000	1.43
Automated Reporting Tool	$45,000	$70,000	1.56
Supply Chain Optimization Model	$75,000	$125,000	1.13
Anomaly Detection in Real-time Data	$60,000	$95,000	1.14

Table 8.3 Features aligned with the Advance Machine Learning and AI Capabilities epic

Feature name	First-year cost	Total cost	Priority score
Customer Segmentation Analysis	$40,000	$60,000	1.50
Personalized Marketing Recommendation Engine	$65,000	$110,000	1.46
Churn Prediction Model	$50,000	$85,000	1.64
Text Mining for Sentiment Analysis	$35,000	$55,000	1.63
Customer Lifetime Value Prediction	$50,000	$80,000	1.50
Time Series Forecasting	$55,000	$90,000	1.23
Recommendation System for Cross-selling	$65,000	$110,000	1.50
Automated ML Model Selection and Tuning	$70,000	$115,000	1.43

To summarize, AIMM will prioritize features to achieve strategic objectives efficiently. The selection process involves the following criteria:

- *Objective function*—The goal is to maximize the priority scores of the selected features, ensuring the highest value relative to the effort required without violating any constraints established beforehand.
- *Number of features*—AAIM will prioritize a minimum of 9 features and a maximum of 12 features.
- *Epic balance*—Each epic will receive a commitment of three or four features to ensure balanced progress across all strategic objectives.
- *Budget constraints*
  - *Yearly investment*—The total investment for the selected features will not exceed $700,000 over the next year.
  - *Total investment*—The combined cost of these features will not exceed $1.1 million over their complete lifecycle.

By adhering to these constraints and focusing on maximizing priority scores, AAIM will strategically allocate resources to ensure a balanced and high-impact advancement of its objectives. This approach uses linear programming and constrained optimization to achieve the most optimal results.

8.2 Developing the linear optimization framework

To effectively utilize linear programming for project prioritization, it's crucial to understand the core requirements of a linear programming problem. The key characteristic of linear programming is that all mathematical relationships involved in the problem must be linear. In other words, the objective function and all constraints should be expressed as linear equations or inequalities.

The core requirements are as follows:

1. *Linear relationships*—The function that we aim to maximize or minimize must be a linear combination of the decision variables. For example, if we are maximizing the priority score, the objective function should look like this:

$$\text{Maximize } Z = c_1x_1 + c_2x_2 + \ldots + c_nx_n$$

where Z is the objective function; c_1, c_2, …, c_n are coefficients representing the value of each feature, represented by the priority score, which is the ratio between the expected value and estimated effort; and x_1, x_2, …, x_n are binary decision variables indicating whether a feature is selected ($x_i = 1$) or not selected ($x_i = 0$).

The constraints must also be linear, meaning each constraint should form a linear equation or inequality. For instance,

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n \leq b$$

where a_1, a_2, …, a_n are coefficients; x_1, x_2, …, x_n are decision variables; and b is a constant representing the limit, or constraint (e.g., maximum investment).

2. *Quantitative objective function*—The objective function is a quantitative measure that the linear programming model aims to optimize. This function could represent maximizing priority scores, minimizing costs, or balancing both aspects, depending on the specific goals of the organization.
3. *Constraints*—Constraints are the restrictions or limitations on the decision variables. These could include budget limits, resource availability, and strategic goals. Each constraint must be expressed in linear form.

Constraints ensure that the solution is feasible and practical within the given limits: for example, ensuring that the total investment does not exceed a specified budget:

$$\sum_{i=1}^{n} \text{cost}_i \times x_i \leq \text{Budget}$$

Strategic constraints might include balancing the number of features selected from each epic to ensure comprehensive progress across all strategic objectives.

4. *Alternatives*—The decision variables (x_1, x_2, …, x_n) represent the alternatives or choices available. The linear programming model evaluates different combinations of these variables to identify the optimal set of features that maximizes the objective function while satisfying all constraints.

By adhering to these requirements, the linear optimization framework ensures that the selected features provide the highest possible value within the given constraints. This structured approach allows AAIM to strategically allocate resources, ensuring a balanced and effective advancement of its objectives.

8.2.1 Explanation of linear equations and inequalities

In the context of linear programming, linear equations and inequalities are fundamental components:

- *Linear equations*—Mathematical expressions in which each term is either a constant or the product of a constant and a single variable. The equation represents a straight line when graphed. For example,

$$3x_1 + 4x_2 = 12$$

This equation states that the sum of 3 times x_1 and 4 times x_2 equals 12. Both x_1 and x_2 are variables, and the coefficients (3 and 4) and the constant term (12) are all linear.

- *Linear inequalities*—These are similar to linear equations but involve inequality signs ($\leq$, $\geq$, $<$, $>$) instead of an equal sign. For instance,

$$5x_1 + 2x_2 \leq 20$$

This inequality means the sum of 5 times x_1 and 2 times x_2 must be less than or equal to 20. Linear equalities represent half-planes when graphed and define the feasible region in a linear programming problem.

In summary, both the objective function and the constraints in a linear programming problem must be expressed using these linear forms to ensure the problem remains solvable using linear programming techniques. This requirement is crucial because it ensures that the relationships between variables are proportional and additive, which is essential for finding the optimal solution.

8.2.2 Data definition

To tackle the challenge of project prioritization, we begin by establishing a robust data source that itemizes the alternatives. The provided data set includes detailed information about each feature, such as the epic it belongs to, the name, the cost for the first

year, the total cost, the value points, and the effort points. This structured data serves as the foundation for our linear optimization framework.

Next, we build out the linear optimization framework by defining the constraints and the objective function and solving the optimization problem. Here is the step-by-step sequence:

1. *Data definition*—We start by defining a list of `Feature` instances, where each instance represents a feature with associated attributes like epic, name, cost for the first year, total cost, value points, and effort points. (Alternatively, a spreadsheet imported as a pandas data frame would work just as well.)
2. *Objective function*—The objective function is designed to maximize the priority score of the selected features. This score is calculated as the ratio of value points to effort points, and we negate the values because the `linprog` method in `scipy.optimize` performs minimization by default.
3. *Constraints*—We establish several constraints to ensure the solution is practical and aligns with organizational goals:
   - *Budget constraints*—Total costs for the first year and overall must not exceed specified limits.
   - *Epic constraints*—Each epic should have at least three and at most four features selected. This ensures balanced progress across all strategic objectives.
4. *Decision variable bounds*—Each feature can either be selected (1) or not selected (0), so the decision variables are binary.
5. *Solving the linear programming problem*—Using the `linprog()` function, we solve the optimization problem to find the optimal set of features that maximizes the priority score while adhering to all constraints.
6. *Result evaluation*—We check whether the solution is successful. If it is, we extract and display the selected features, along with their costs and priority scores. This helps in understanding which features provide the greatest value relative to their effort and cost, ensuring efficient resource allocation.

Our first snippet of Python code uses a data class named `Feature` to define the attributes of each feature in a structured way. We then create an object called `features`, which contains multiple instances of `Feature`. Each instance represents a feature and includes detailed attributes, such as the `epic` it belongs to, the feature `name`, the first-year cost (`cost_1yr`), the total cost (`total_cost`), the value points (`value_points`), and the effort points (`effort_points`). Organized by their respective epics, these feature instances allow for clear categorization, making it easy to analyze and prioritize features based on strategic objectives.

This data class structure enhances readability and enables efficient data manipulation, supporting optimization and decision-making processes. Note that the subsequent code requires the `dataclass` decorator, which is imported from the `dataclasses` module, to define the `Feature` class. For brevity, only the first four and last four features are shown here. These instances correspond to tables 8.1–8.3, with

the distinction that the code displays value and effort points separately, whereas the tables calculate a priority score based on these variables:

```
>>> from dataclasses import dataclass
>>> @dataclass
>>> class Feature:
>>>     epic: str
>>>     name: str
>>>     cost_1yr: int
>>>     total_cost: int
>>>     value_points: int
>>>     effort_points: int
>>> features = [
>>>     Feature('Enhance Data Infrastructure',
>>>        'Automated Data Cleaning Pipeline',
>>>             50000, 80000, 16, 13),
>>>     Feature('Enhance Data Infrastructure',
>>>             'Real-time Data Ingestion',
>>>             70000, 120000, 18, 14),
>>>     Feature('Enhance Data Infrastructure',
>>>             'Data Privacy and Compliance Tools',
>>>             80000, 130000, 19, 16),
>>>     Feature('Enhance Data Infrastructure',
>>>             'Real-time Data Monitoring and Alerts',
>>>             60000, 100000, 18, 13),

>>> # Other feature details not included
>>> # for space considerations

>>>     Feature('Advance Machine Learning and AI Capabilities',
>>>             'Customer Lifetime Value Prediction',
>>>             50000, 80000, 15, 10),
>>>     Feature('Advance Machine Learning and AI Capabilities',
>>>             'Time Series Forecasting',
>>>             55000, 90000, 16, 13),
>>>     Feature('Advance Machine Learning and AI Capabilities',
>>>             'Recommendation System for Cross-selling',
>>>             65000, 110000, 18, 12),
>>>     Feature('Advance Machine Learning and AI Capabilities',
>>>        'Automated ML Model Selection and Tuning',
>>>             70000, 115000, 20, 14)
>>> ]
```

Automatically adds useful methods to the Feature class for initializing attributes and providing readable string representations

Defines the Feature class

Epic category, as a string

Feature name, as a string

First-year cost, as an integer

Total cost, as an integer

Value points score, as an integer

Effort points score, as an integer

Epic for the first of 20 features

Feature name

First-year cost, total, cost, value points score, and effort points score

Code comment ignored by the interpreter at runtime; always preceded by a pound sign

Epic for the last of 20 features

Feature name

First-year cost, total, cost, value points score, and effort points score

Now that we have clearly defined our data source, we can proceed to the next step: demonstrating how to define the objective function. This crucial component will allow us to effectively prioritize and select features based on our established criteria, ensuring that we maximize value while adhering to our constraints.

8.2.3 Objective function

The following line of code sets up the objective function for our optimization problem. By iterating through each feature in the `features` list, it calculates a priority

score as the ratio of value points to effort points for each feature. The negative sign is applied to each calculated priority score to transform the problem, allowing a minimization algorithm to effectively maximize this value-to-effort ratio. This transformation ensures that the optimization solver will prioritize features that offer the highest value relative to their effort:

```
>>> c = [-f.value_points / f.effort_points for f in features]
```

It produces what is known as a Python list comprehension. List comprehensions are a concise way to create lists in Python, offering a syntactically compact and readable method to build lists from existing lists or iterables. In this context, the list comprehension iterates over each feature in the `features` list. Each `Feature` instance represents a specific feature with various attributes such as `value_points` and `effort_points`.

The expression `f.value_points / f.effort_points` calculates the priority score for each feature `f`. This score is a ratio indicating the value derived per unit of effort, where a higher ratio suggests that the feature provides more value for the effort required.

The negative sign (`-`) before the expression is used because linear programming solvers like `linprog()` typically minimize the objective function by default. By negating the priority scores, we effectively transform the problem of maximizing priority scores into a minimization problem, which the solver can handle directly.

The result of the list comprehension is a list of these negated priority scores, which will be used as coefficients in the objective function for the linear programming solver. The `print()` method returns the derived priority scores, rounded to two digits right of the decimal point:

```
>>> print([round(val, 2) for val in c])
[-1.23, -1.29, -1.19, -1.38, -1.42, -1.45, -1.67, -1.13, -1.43, -1.56,
-1.13, -1.14, -1.5, -1.46, -1.64, -1.62, -1.5, -1.23, -1.5, -1.43]
```

Aside from the negation of the priority scores for each feature, these scores align perfectly with those in tables 8.1–8.3.

8.2.4 Constraints

Our next snippet of code defines the inequality constraints for a linear programming problem and returns the constraints as NumPy arrays: `cost_1yr` must be ≤ to `700000`, and `total_cost` must be ≤ to `1100000`. In other words, AIMM has just enough resources to invest up to $700,000 in the coming year and no more than $1.1 million in total for the selected features:

```
>>> import numpy as np
>>> A_ub = np.array([
>>>     [f.cost_1yr for f in features],
>>>     [f.total_cost for f in features]
>>> ])
>>> b_ub = np.array([700000, 1100000])
```

The variable `A_ub` is a matrix that represents the coefficients of the inequality constraints for the linear programming problem. In this case, it is constructed as a NumPy array, using list comprehensions that iterate over the `features` list. One list comprehension extracts the `cost_1yr` (cost in the first year) for each feature in the `features` list, thereby creating an array of the first-year costs of all features. This array becomes the first row of the `A_ub` matrix, representing the coefficients for the first inequality constraint related to the first-year budget.

Similarly, another list comprehension extracts the `total_cost` for each feature in the `features` list, creating an array of the total costs of all features. This list becomes the second row of the `A_ub` matrix, representing the coefficients for the second inequality constraint related to the total allowable budget.

A call to the `print()` method returns the `A_ub` matrix, now represented as a NumPy matrix:

```
>>> print(A_ub)
[[ 50000  70000  80000  60000  50000  55000  60000  55000  80000
   45000  75000  60000  40000  65000  50000  35000  50000  55000
   65000  70000]
 [ 80000 120000 130000 100000  85000  90000 100000  90000 130000
   70000 125000  95000  60000 110000  85000  55000  80000  90000
   110000 115000]]
```

End of the first row in the array. Figures represent the estimated first-year cost for each feature.

End of the second row in the array. Figures represent the estimated total cost for each feature.

The variable `b_ub` is a vector that represents the right-hand side values of the inequality constraints defined by `A_ub`. Each element in `b_ub` corresponds to a specific constraint:

- `7000000` represents the maximum allowable budget for the first year. The first inequality constraint ensures that the sum of the selected features' first-year costs does not exceed this amount.
- `1100000` represents the maximum allowable total budget for the entire project duration. The second inequality constraint ensures that the sum of the selected features' total costs does not exceed this amount.

Together, `A_ub` and `b_ub` define the set of linear inequality constraints for the optimization problem. These constraints ensure that the selected features do not exceed the specified budget limits, both for the first year and overall. When used in conjunction with the optimization solver, these constraints help in finding the optimal set of features that maximizes the objective function (priority scores) while adhering to the budgetary limits.

Beyond budgetary constraints, AAIM wants to pursue a mix of features that enable it to balance its investment across different epics or strategic objectives. This approach ensures that its resources are not only optimally utilized within financial limits but also strategically distributed to support diverse areas of development. By balancing investments across epics, AAIM can ensure comprehensive progress in enhancing data

infrastructure, improving data analytics and reporting, and advancing machine learning and AI capabilities.

The following code snippet introduces additional constraints to achieve this balance. It iterates over each epic, creating constraints to ensure that at least three and no more than four features are selected from each epic. For each epic, two rows are constructed and appended to the `A_ub` matrix: one representing the lower bound (at least three features) and the other representing the upper bound (no more than four features). The corresponding values are added to the `b_ub` vector, completing the inequality constraints necessary for the linear programming model:

```
>>> epics = ['Enhance Data Infrastructure',
>>>          'Improve Data Analytics and Reporting',
>>>          'Advance Machine Learning and AI Capabilities']

>>> for epic in epics:
>>>     row_min = [-1 if f.epic == epic else 0 for f in features]
>>>     row_max = [1 if f.epic == epic else 0 for f in features]
>>>     A_ub = np.vstack([A_ub, row_min])
>>>     A_ub = np.vstack([A_ub, row_max])
>>>     b_ub = np.append(b_ub, -3)
>>>     b_ub = np.append(b_ub, 4)
```

The preceding code can best be explained by breaking it down into the following parts:

- `epics = ['Enhance Data Infrastructure',...` initializes a list to define the epics that will be iterated over to create the constraints.
- `for epic in epics` loops through each epic in the `epics` list to create and add constraints for that epic.
- `row_min...` creates a row for the `A_ub` matrix using a list comprehension. Each element in the row is `-1` if the feature belongs to the current epic or `0` otherwise. Negative coefficients are used to set up a "greater than or equal to" inequality for the minimum constraint.
- Similarly, `row_max...` creates another row for the `A_ub` matrix using a list comprehension. Each element is `1` if the feature belongs to the current epic or `0` if otherwise. The positive coefficients are used to set up a "less than or equal to" inequality for the maximum constraint `row_min`, which is appended to the `A_ub` matrix.
- `A_ub = np.vstack...` appends the `row_min` array to the `A_ub` matrix using the NumPy `vstack()` function, ensuring consistency with its array structure.
- `b_ub = np.append...` appends `-3` to the `b_ub` vector using the NumPy `append()` function. This enforces that the sum of selected features for the current epic is at least `3` (since the inequality `-sum ≥ -3` translates to `sum ≤ 3`).
- `A_ub = np.vstack...` appends the `row_max` array to the `A_ub` matrix, adding the coefficients for the "at most `4`" constraint.

- `b_ub = np.append...` appends `4` to the `b_ub` vector, representing the right-hand side of the inequality. This enforces that the sum of selected features for the current epic is at most `4`.

The last bit of code ensures that the selected features are balanced across the three epics by adding constraints to the linear programming model. Specifically, it guarantees that each epic has at least three and at most four features selected. This is done by constructing and appending rows to the `A_ub` matrix and corresponding values to the `b_ub` vector, which represent the inequality constraints for the optimization problem.

8.2.5 Decision variable bounds

Our next code snippet is an integral part of setting up a linear programming problem, specifically in defining the bounds for the decision variables. In linear programming, decision variables represent the choices to be made—in this case, whether to select a feature or not. This particular line of code utilizes a list comprehension to create a list of tuples, where each tuple represents the bounds for a corresponding decision variable. Because each decision variable must be either `0` or `1`, this setup represents a binary integer programming problem—a specialized form of linear programming with discrete, yes-or-no choices:

```
>>> x_bounds = [(0, 1) for _ in features]
```

In the context of our problem, each feature can be either selected or not, which corresponds to a binary decision variable. The values for these decision variables can only be `0` or `1`, where `0` indicates the feature is not selected and `1` indicates it is selected. The line of code iterates over the list of features, creating a tuple `(0, 1)` for each feature. This tuple is the bound for the decision variable associated with that feature.

By defining `x_bounds` in this manner, we ensure that the optimization solver respects these bounds when determining the optimal solution. Essentially, this constraint enforces that the solution comprises only binary decisions, aligning with the reality of project selection, where a feature can either be included or excluded.

This list of tuples, `x_bounds`, is then passed to the solver along with other parameters like the objective function and constraints. The solver uses this information to explore feasible solutions within the specified bounds and find the optimal set of features that maximize the priority scores while adhering to budgetary and strategic constraints.

8.2.6 Solving the linear programming problem

So far, we have laid the groundwork for solving a project prioritization problem using linear programming. We began by defining a data set of features, each belonging to specific epics and characterized by attributes such as cost, value points, and effort points. Using this data set, we constructed the objective function, aimed at maximizing the priority scores of the selected features. We also established budgetary constraints, ensuring that the total cost for the first year and the overall cost stay within specified limits.

Additionally, we introduced constraints to balance the number of selected features across different epics. This ensures that resources are evenly distributed across strategic objectives, with at least three and no more than four features chosen from each epic. We then defined the bounds for our decision variables, ensuring they are binary (0 or 1), representing whether a feature is selected or not.

The following code uses the `scipy` library to solve the linear programming problem:

```
>>> from scipy.optimize import linprog
>>> import numpy as np
>>> result = linprog(c, A_ub = A_ub, b_ub = b_ub,
>>>                  bounds = x_bounds, method = 'highs')
```

The first line imports the `linprog` method from the `optimize` module of the `scipy` library. The `linprog()` method is used to solve linear programming problems, enabling us to find the optimal solution that maximizes or minimizes a specified objective function while adhering to a set of linear constraints.

It achieves this by taking in the following parameters:

- `c` is the coefficient list for the objective function, which we previously defined to represent the negated priority scores of the features. By minimizing this objective function, we effectively maximize the priority scores.
- `A_ub` is the matrix representing the coefficients of the inequality constraints. Each row in `A_ub` corresponds to a constraint, and each column corresponds to a decision variable (feature). The entries in `A_ub` specify how each decision variable contributes to each constraint.
- `b_ub` is the vector containing the right-hand-side values for the inequality constraints. Each entry in `b_ub` corresponds to a constraint, defining the upper bound that the linear combination of decision variables (weighted by `A_ub`) must not exceed.
- `bounds` is a parameter specifying the bounds for each decision variable. In our case, we defined `x_bounds` as `0` or `1`, thereby indicating that each decision variable must represent whether or not a feature is selected.
- `method` parameter determines the algorithm used to solve the linear programming problem. The `'highs'` method selects and applies the most appropriate algorithm, such as `'simplex'` or `'interior-point,'` depending on the problem's size and complexity.

The `result` variable stores the output from the `linprog()` method. This output includes three important attributes that can be returned by calling the `print()` method. The first, `result.success`, is a Boolean—that is, a data type that can have just one of two possible values, `True` or `False`—indicating whether or not the optimization was successful:

```
>>> print(result.success)
True
```

When `result.success` is printed and returns `False`, it indicates that the optimization problem could not be solved with the given constraints. This could be due to several reasons, such as the constraints being too restrictive, thereby making the problem infeasible, or the problem not having a bounded solution within the specified constraints.

The second attribute, `result.x`, is an array that contains the values of the decision variables that optimize the objective function. These values indicate which features are selected (with values of `1`) and which are not (with values of `0`) in the context of a binary decision problem. Essentially, `result.x` provides the optimal solution to the linear programming problem, showing how the resources should be allocated to maximize the priority scores while satisfying all the constraints:

```
>>> print(result.x)
```

```
[ 0. 1. 0. 1. 1. 1. 1. 0. 1. 1. 0. 1. 1. 0. 1. 1. 1. 0.
 -0. 0.]
```

So, for example, the first feature listed in our data source, Automated Data Cleaning Pipeline, should not be selected, but the second feature, Real-time Data Ingestion, should be.

Overall, the maximum of 12 features were selected, as determined by summing the values in the `result.x` array:

```
>>> print(sum(result.x))
12.0
```

The third attribute, `result.fun`, is the value of the objective function at the optimal solution. This value represents the maximum (or minimum) priority score achieved by the selected features, adjusted for the negation used in the setup. By evaluating `result.fun`, we can determine the effectiveness of our feature selection in terms of maximizing the priority score while adhering to all constraints. This provides a quantitative measure of the optimal solution's overall value.

Our next call to the `print()` method returns the sum of the priority scores from the selected features. This total is multiplied by `-1` to reverse the negation applied during setup and then rounded to include no more than two digits to the right of the decimal point:

```
>>> print(round(result.fun * -1, 2))
17.6
```

When the sum of the priority scores (17.6) is divided by the number of allowed features (12), we get an average priority score equal to 1.47.

By calling the `linprog()` method with our defined parameters, we solve the project prioritization problem, identifying the optimal set of features that maximize the priority scores while adhering to budgetary and strategic constraints. This approach ensures that AAIM can make informed, strategic decisions about resource allocation, ultimately driving efficient and balanced progress across its key objectives.

8.2.7 Result evaluation

Given the success of our linear optimization problem, we can confidently state that AAIM can invest in up to 12 features without exceeding its budgetary constraints for this year or next. This means AAIM can prioritize 12 of the features it shortlisted from the backlog. Although it is possible to manually derive these features by matching the `result.x` array with our features list, this approach is cumbersome and inefficient.

Fortunately, there is a more efficient method to obtain the full result set, which includes the list of selected features as well as the final budget numbers for both the current and following year. By using the attributes provided in the optimization result, we can easily extract and present this information. The selected features can be identified directly from the `result.x` array, where a value of `1` indicates selection. Additionally, the final budget numbers can be computed by summing the relevant cost attributes of the selected features. This method simplifies the process while ensuring accurate and complete reporting of the optimization outcomes.

Here's one example of how this can be achieved this programmatically:

```
>>> if result.success:
>>>     selected_features = [f for f, \
>>>        x in zip(features, result.x) \
>>>                             if x == 1]
>>>     print('Selected Features:')
>>>     for f in selected_features:
>>>         print(f'Epic: {f.epic}, '
>>>               f'Feature: {f.name}, '
>>>               f'Cost in 1 Year: ${f.cost_1yr}, '
>>>               f'Total Cost: ${f.total_cost}, '
>>>               f'Priority Score: {f.value_points /
>>>                   f.effort_points:.2f}')
>>>     total_cost_1yr = \
>>>         sum(f.cost_1yr for f in selected_features)
>>>     total_cost = \
>>>         sum(f.total_cost for f in selected_features)
>>>     print(f'\nTotal Cost in 1 Year: $' \
>>>     f'{total_cost_1yr}')
>>>     print(f'Total Cost: ${total_cost}')
>>> else:
>>>     print('Optimization failed.')

Selected Features:
Epic: Enhance Data Infrastructure,
Feature: Real-time Data Ingestion,
Cost in 1 Year: $70000, Total Cost: $120000, Priority Score: 1.29

Epic: Enhance Data Infrastructure,
Feature: Real-time Data Monitoring and Alerts,
Cost in 1 Year: $60000, Total Cost: $100000, Priority Score: 1.38

Epic: Enhance Data Infrastructure,
Feature: Integration with External Data Sources,
```

Checks whether the optimization problem is successfully solved

Creates a list of features that were selected in the optimization solution

Prints the header

Initiates a loop to iterate over each selected feature

Prints the epic of the current feature in the loop

Prints the feature name

Prints the feature cost for the current year

Prints the total feature cost

Derives and prints the priority score

Calculates the total cost for the first year for all selected features

Calculates the total overall cost for all selected features

Prints the total overall cost

Prints the total cost for the first year

Prints "Optimization failed" if the optimization was not successful

```
Cost in 1 Year: $50000, Total Cost: $85000, Priority Score: 1.42

Epic: Enhance Data Infrastructure,
Feature: Feature Engineering Automation,
Cost in 1 Year: $55000, Total Cost: $90000, Priority Score: 1.45

Epic: Improve Data Analytics and Reporting,
Feature: Interactive Data Visualization Dashboard,
Cost in 1 Year: $60000, Total Cost: $100000, Priority Score: 1.67

Epic: Improve Data Analytics and Reporting,
Feature: Fraud Detection System,
Cost in 1 Year: $80000, Total Cost: $130000, Priority Score: 1.43

Epic: Improve Data Analytics and Reporting,
Feature: Automated Reporting Tool,
Cost in 1 Year: $45000, Total Cost: $70000, Priority Score: 1.56

Epic: Improve Data Analytics and Reporting,
Feature: Anomaly Detection in Real-time Data,
Cost in 1 Year: $60000, Total Cost: $95000, Priority Score: 1.14

Epic: Advance Machine Learning and AI Capabilities,
Feature: Customer Segmentation Analysis,
Cost in 1 Year: $40000, Total Cost: $60000, Priority Score: 1.50

Epic: Advance Machine Learning and AI Capabilities,
Feature: Churn Prediction Model,
Cost in 1 Year: $50000, Total Cost: $85000, Priority Score: 1.64

Epic: Advance Machine Learning and AI Capabilities,
Feature: Text Mining for Sentiment Analysis,
Cost in 1 Year: $35000, Total Cost: $55000, Priority Score: 1.62

Epic: Advance Machine Learning and AI Capabilities,
Feature: Customer Lifetime Value Prediction,
Cost in 1 Year: $50000, Total Cost: $80000, Priority Score: 1.50

Total Cost in 1 Year: $655000
Total Cost: $1070000
```

By using this approach, AAIM can quickly and conveniently obtain a comprehensive view of the selected features and their associated costs, ensuring that all strategic and budgetary goals are met. This method not only makes the process more efficient but also provides a clear and detailed summary of the optimization results.

One of the advantages of this framework is its flexibility in accommodating various "what-if" scenarios by simply adjusting the parameters. For example, we can explore the effect of changing the constraints to require a minimum of two and a maximum of three features per epic, or setting different budget limits, such as a $550,000 investment in the first year and a total of $850,000. This adaptability allows AAIM to quickly assess different strategic options and make informed decisions based on changing priorities or resource availability without necessitating a complete code rewrite.

In the next chapter, we will transition from linear programming to another powerful decision-making technique: Monte Carlo simulations. This approach allows us to model and analyze uncertainty and variability in complex systems, providing valuable insights for informed decision-making and effective risk management.

Summary

- Linear programming is a crucial technique for efficiently allocating limited resources, such as budget, time, and materials, to achieve the best possible outcomes. It ensures that resources are utilized in the most effective way to maximize benefits or minimize costs.
- A fundamental aspect of linear programming is that both the objective function and the constraints must be linear. This means all relationships in the model are proportional and additive, making the mathematical formulation simpler and solvable using efficient algorithms.
- The foundation of linear programming involves defining an objective function that needs to be either maximized or minimized. This function is subject to a set of linear constraints, which represent the real-world limitations and requirements, such as budget caps, resource availability, and minimum performance standards.
- One of the major strengths of linear programming is its flexibility. It allows for easy adjustments to the parameters and constraints, enabling the exploration of various "what-if" scenarios. This adaptability helps decision-makers understand the potential effects of different strategic choices and changes in resource availability.
- Linear programming can accommodate both binary decision variables, which represent yes/no decisions (e.g., whether to undertake a project or not), and continuous decision variables, which represent quantities that can take any value within a given range. This versatility makes it suitable for a wide range of optimization problems.
- By incorporating multiple constraints, linear programming helps ensure that decisions are balanced and strategically sound. It supports comprehensive decision-making by allowing for the consideration of various objectives and priorities, ensuring that all strategic goals are advanced in a harmonious manner.