Modul M323 – Funktionales Programmieren

Bank Marketing Data

Kundenanalyse & Abschlussprognose

Projekt-Dokumentation

Autoren: Peter Ngo, Alex Uscata

Klasse: INA 23A

Dozent: Dieter Kopp

Datum: 11. Januar 2026

Inhaltsverzeichnis

1 Einleitung

Im Rahmen des Moduls M323 – Funktionales Programmieren wird in diesem Projekt eine Datenanalyse einer Bank-Marketingkampagne umgesetzt. Ziel ist es, eine imperativ geschriebene Lösung mit einer funktional refaktorierten Version zu vergleichen und die Vor- sowie Nachteile funktionaler Sprachelemente praxisnah zu evaluieren.

Als Datengrundlage dient ein realitätsnaher Datensatz einer portugiesischen Bank, welcher Kundendaten sowie den Erfolg einer Marketingkampagne enthält. Beide Programmversionen erzeugen denselben Output, unterscheiden sich jedoch grundlegend in der Programmierweise.

Die verwendeten Daten werden in Abschnitt 3 beschrieben. Der Einfluss funktionaler Sprachelemente wird anschließend im Refactoring-Vergleich in Abschnitt 6 analysiert.

2 Wahl der imperativen Programmiersprache

2.1 Begründung der Sprachwahl

Für dieses Projekt wurde die Programmiersprache **Python** gewählt. Python eignet sich besonders gut für Datenanalysen, da die Sprache:

- leicht lesbar und verständlich ist
- imperative Programmierung vollständig unterstützt
- funktionale Sprachelemente integriert anbietet
- häufig im beruflichen Umfeld eingesetzt wird

Da Python sowohl imperative als auch funktionale Konzepte vereint, ist die Sprache ideal geeignet, um beide Programmierparadigmen direkt miteinander zu vergleichen.

2.2 Unterstützte funktionale Elemente in Python

Python ist keine rein funktionale Sprache, bietet jedoch zahlreiche funktionale Sprachelemente aus der Standardbibliothek:

- `map()` – Transformation von Daten
- `filter()` – Selektion von Elementen
- `reduce()` (aus `functools`) – Aggregation
- Lambda-Funktionen
- List Comprehensions
- Unveränderliche Datentypen (z. B. Tupel)

Diese Elemente ermöglichen eine deklarative und kompakte Beschreibung von Datenverarbeitungslogik.

2.3 Funktionale Sprachelemente in Python – Detailanalyse

Neben den grundlegenden funktionalen Sprachelementen weist Python im Vergleich zu rein funktionalen Sprachen sowohl konzeptionelle Stärken als auch Einschränkungen auf.

Higher-Order Functions Python unterstützt Higher-Order Functions nativ, allerdings mit einigen charakteristischen Besonderheiten:

- `map()` und `filter()` liefern Iteratoren und arbeiten damit lazily. Dies spart Speicher, verhindert jedoch ein mehrfaches Durchlaufen derselben Daten ohne explizite Materialisierung (z. B. mittels `list()`).
- `reduce()` wurde seit Python 3 in das Modul `functools` ausgelagert. Diese Designentscheidung reflektiert, dass in Python explizite Schleifen oder spezielle Aggregatfunktionen (`sum`, `min`, `max`) häufig als besser lesbar angesehen werden.

Im Projekt wird `reduce()` dennoch bewusst eingesetzt, um die funktionale Aggregation explizit sichtbar zu machen (z. B. bei der Gruppierung von Datensätzen oder der Berechnung von Buckets).

Limitierungen gegenüber Haskell Im Vergleich zu rein funktionalen Sprachen wie Haskell weist Python mehrere Einschränkungen auf:

- Keine Tail-Call-Optimierung: Rekursive Funktionen sind durch eine begrenzte Rekursionstiefe (ca. 1000 Aufrufe) eingeschränkt und daher für viele funktionale Muster ungeeignet.
- Mutable Default Arguments können unbeabsichtigte Seiteneffekte verursachen und widersprechen dem Prinzip der funktionalen Reinheit.
- Kein echtes Pattern Matching auf Funktionsebene; erst ab Python 3.10 steht mit `match/case` ein strukturelles Matching für Kontrollflüsse zur Verfügung, das jedoch nicht die Ausdrucksstärke funktionaler Pattern Matches erreicht.

Damit wird deutlich, dass Python funktionale Konzepte eher als ergänzende Sprachmittel bereitstellt, nicht jedoch als tragendes Paradigma erzwingt. Die funktionale Programmierung in Python beruht daher stärker auf Programmierdisziplin als auf sprachlicher Absicherung durch den Compiler.

2.3.1 Kurzbeispiele der funktionalen Elemente

```python
# map: Transformation
list(map(lambda x: x*x, [1,2,3]))       # -> [1,4,9]

# filter: Selektion
list(filter(lambda x: x > 2, [1,2,3])) # -> [3]

# reduce: Aggregation
from functools import reduce
reduce(lambda acc, x: acc + x, [1,2,3], 0)   # -> 6
```

Diese funktionalen Konzepte werden im Projekt konkret eingesetzt für:

- Filterung von Datensätzen über Prädikatfunktionen (`apply_filters`)
- Gruppierung von Daten mittels `reduce` (`group_by_key`)
- Numerische Transformationen und Aggregationen (`mean`, ANOVA)

3 Datengrundlage

3.1 Beschreibung des Datensatzes

Als Datengrundlage dient ein Bank-Marketing-Datensatz einer portugiesischen Bank, der auf einer öffentlich dokumentierten Bank-Marketing-Studie basiert (vgl. [1]). Der Datensatz enthält Kundendaten aus einer Marketingkampagne sowie eine Zielvariable, welche angibt, ob ein Produktabschluss erfolgt ist (`complete = yes/no`).

Die Daten liegen in Form einer CSV-Datei vor und wurden im Projekt ausschließlich lesend verwendet.

3.2 Hauptdimensionen

Der Datensatz lässt sich in folgende Hauptdimensionen einteilen:

- **Demografie:** Alter (`age`), Bildungsstand (`education`), Familienstand (`marital`)
- **Finanzen:** Kontostand (`balance`)
- **Kontaktdaten:** Gesprächsdauer (`duration`)
- **Ergebnisvariable:** Produktabschluss (`complete`)

3.3 Verwendete Variablen

Im Projekt werden unter anderem folgende Variablen verwendet:

- `age` (numerisch)
- `job` (kategorisch)
- `marital` (kategorisch)
- `education` (kategorisch)
- `balance` (numerisch)
- `housing` (binär)
- `loan` (binär)
- `duration` (numerisch)
- `complete` (binär, Zielvariable)

3.4 Herkunft des Datensatzes und Abgrenzung

Der im Projekt verwendete Bank-Marketing-Datensatz stammt ursprünglich aus einer universitären Lehrveranstaltung zur multivariaten Datenanalyse an der Universität Basel (vgl. [2]). Der Datensatz wurde im Rahmen eines Assignments zur statistischen Auswertung von Bank-Marketingkampagnen eingesetzt und basiert auf realitätsnahen, vorverarbeiteten Kundendaten einer portugiesischen Bank.

Die Aufgabenstellung des ursprünglichen Assignments umfasste unter anderem Varianzanalysen, logaritmische Transformationen sowie Klassifikationsverfahren. In diesem Projekt wird der Datensatz jedoch ausschließlich als Datengrundlage verwendet und in einen neuen Kontext übertragen.

Die Zielsetzung dieses Projekts unterscheidet sich bewusst von der ursprünglichen universitären Aufgabenstellung: Der Fokus liegt nicht auf statistischer Modellbildung oder Inferenz, sondern auf dem Vergleich imperativer und funktionaler Programmierparadigmen im Sinne des Moduls M323 – Funktionales Programmieren.

4 Projektantrag

4.1 Ausgangslage

Während einer Marketingkampagne hat eine Bank verschiedene Kundendaten erfasst. Der Datensatz liegt in Form einer CSV-Datei vor und ist nicht direkt für Analysen aufbereitet.

Ziel ist es, relevante Kennzahlen zu berechnen, Kundengruppen zu vergleichen und statistische Zusammenhänge zu untersuchen.

4.2 Zielsetzung

Das Projekt verfolgt folgende Ziele:

- Analyse der Erfolgsquote der Marketingkampagne
- Vergleich verschiedener Kundengruppen
- Umsetzung einer imperativen Version (V1.0)
- Refactoring in eine funktionale Version (V2.0)
- Vergleich beider Ansätze hinsichtlich Lesbarkeit und Wartbarkeit

4.3 Projektumfang

Das Projekt umfasst:

- Einlesen und Verarbeiten eines CSV-Datensatzes
- Textbasierte Konsolenausgabe
- Keine externen Bibliotheken

- Umsetzung in Python; Ausführung und Dokumentation der Experimente in einem Jupyter Notebook (Tooling).

5 Programmausgabe

Beide Versionen des Programms erzeugen identische Resultate für die zentralen Produktfunktionen und Kernanalysen. Die Ausgabe erfolgt textbasiert in der Konsole.

5.1 Beispielhafter Output

Das folgende Beispiel zeigt die Konsolenausgabe der Auswertungsfunktion *Group by Education* (Menüoption 5). Dabei werden alle Datensätze nach Bildungsniveau gruppiert und aggregierte Kennzahlen berechnet, darunter die Anzahl der Kunden, das durchschnittliche Alter, der durchschnittliche Kontostand sowie die Erfolgsquote der Marketingkampagne.

Die dargestellte Ausgabe wird sowohl von Version 1.0 (imperative Umsetzung) als auch von Version 2.0 (funktionale Umsetzung) identisch erzeugt.

```
                          GROUP BY EDUCATION
======================================================================
education | count | avg(age) | avg(balance) | success
----------+-------+----------+--------------+--------
primary   |  1002 |     49.0 |      1970.88 |   48.3%
secondary |  3815 |     40.7 |      1709.54 |   52.2%
tertiary  |  2657 |     39.7 |      2404.61 |   62.5%

ANOVA F-Wert: 41.93 (df=2,7471)
```

Hinweis: Version 1.0 (imperativ) und Version 2.0 (funktional) liefern identische Resultate. Weitere exemplarische Konsolenausgaben zu anderen Menüoptionen befinden sich im Appendix (vgl. Abschnitt 11.2).

5.2 Validierung der Resultate

Die automatisierten Tests befinden sich im Verzeichnis `tests/` und können direkt über die Kommandozeile ausgeführt werden:

```
python tests/test_outputs.py
```

Zur Validierung wurden mehrere automatisierte Tests implementiert, welche die funktionale und die imperative Implementierung systematisch miteinander vergleichen. Ziel dieser Tests ist es, sicherzustellen, dass beide Programmversionen trotz unterschiedlicher Programmierparadigmen identische inhaltliche Resultate liefern.

Konkret wurden folgende Aspekte überprüft:

- Vergleich der Gruppierungsresultate (*Group by Education*): Gruppennamen, Gruppengrößen, durchschnittliches Alter, durchschnittlicher Kontostand sowie Erfolgsquoten stimmen

in beiden Implementierungen überein.

- Vergleich des berechneten ANOVA-F-Werts inklusive der Freiheitsgrade ($df_{between}$ und df_{within}).
- Invarianzprüfung: Die Summe der Gruppengrößen entspricht stets der Gesamtanzahl der Datensätze, und alle Erfolgsquoten liegen im gültigen Wertebereich $[0, 1]$.
- Determinismus: Mehrfache Ausführungen mit identischen Eingabedaten liefern in beiden Implementierungen stets exakt dieselben Resultate.
- Behandlung von Randfällen: Für unzureichende Datengrundlagen (z. B. leere Datensätze oder nur eine vorhandene Gruppe) liefert die ANOVA-Funktion korrekt den Wert `None`.

Alle Tests wurden erfolgreich ausgeführt und bestätigen die inhaltliche Äquivalenz der funktionalen und imperativen Version. Damit ist sichergestellt, dass die Unterschiede zwischen beiden Programmen ausschließlich in der Programmstruktur und im verwendeten Paradigma liegen, nicht jedoch in den berechneten Ergebnissen.

5.3 Interpretation und statistische Einordnung

Die Ergebnisse zeigen, dass der durchschnittliche Kontostand sowie die Erfolgsquote mit steigendem Bildungsniveau zunehmen. Kunden mit tertiärer Ausbildung weisen sowohl den höchsten durchschnittlichen Kontostand als auch die höchste Erfolgsquote auf.

Zur Bewertung der Unterschiede zwischen den Bildungsgruppen wird eine einfaktorielle Varianzanalyse (ANOVA) verwendet. Der ANOVA-F-Wert beschreibt das Verhältnis der Varianz zwischen den Gruppen zur Varianz innerhalb der Gruppen. Ein hoher F-Wert deutet darauf hin, dass sich die Mittelwerte der Gruppen deutlich voneinander unterscheiden und die Varianz zwischen den Bildungsgruppen größer ist als die Varianz innerhalb der Gruppen.

Die angegebenen Freiheitsgrade (df) geben an, wie viele unabhängige Informationen in die Berechnung eingeflossen sind. Dabei beschreibt $df_{between}$ die Anzahl der verglichenen Gruppen minus eins, während df_{within} die Streuung der Werte innerhalb der Gruppen widerspiegelt [3].

Die Varianzanalyse wird in diesem Projekt ausschließlich deskriptiv verwendet. Der F-Wert dient hier als Maßzahl zur Beschreibung der Stärke von Unterschieden zwischen den Gruppen und nicht als formale Entscheidungsregel im Sinne eines statistischen Signifikanztests. Auf die Berechnung von p-Werten und eine formale Hypothesenprüfung wird daher bewusst verzichtet.

6 Refactoring: Vergleich Version 1.0 vs. Version 2.0

Beide Programmversionen implementieren dieselben Produktfunktionen und erzeugen identische Resultate, unterscheiden sich jedoch in der Umsetzung: Version 1.0 verwendet primär imperative Sprachmittel (Schleifen, Hilfsvariablen, schrittweiser Aufbau von Resultaten), während Version 2.0 die gleiche Logik mit funktionalen Sprachmitteln (z. B. `map`, `filter`, `reduce`, Lambdas) ausdrückt.

6.1 Konkrete Beispiele aus dem Projekt

6.1.1 Beispiel A: Filtern von Datensätzen

In Version 1.0 wird die Filterlogik mit einer Schleife und `continue` umgesetzt. In Version 2.0 wird dieselbe Logik deklarativ als Prädikatfunktion formuliert und per `filter` angewendet.

Imperative Umsetzung (Version 1.0)

Listing 1: Imperative Filterung mittels Schleifen und Kontrollfluss aus `imperative_version.py`

```
68
69 def apply_filters(data: List[Dict[str, Any]], housing: Optional[bool],
       loan: Optional[bool], balance_gt: Optional[float]) -> List[Dict[str,
       Any]]:
70
71     out: List[Dict[str, Any]] = []
72     for row in data:
73         if housing is not None and row.get("housing") is not housing:
74             continue
75         if loan is not None and row.get("loan") is not loan:
76             continue
77         bal = row.get("balance")
78         if balance_gt is not None:
79             if bal is None or bal <= balance_gt:
80                 continue
81         out.append(row)
82     return out
```

Funktionale Umsetzung (Version 2.0)

Listing 2: Funktionale Filterung mittels Prädikatfunktion und `filter` aus `functional_version.py`

```
139
140 def apply_filters(data: List[Dict[str, Any]], housing: Optional[bool],
       loan: Optional[bool], balance_gt: Optional[float]) -> List[Dict[str,
       Any]]:
141
142     def pred(row: Dict[str, Any]) -> bool:
143         if housing is not None and row.get("housing") is not housing:
144             return False
145         if loan is not None and row.get("loan") is not loan:
146             return False
147         if balance_gt is not None:
148             bal = row.get("balance")
149             if bal is None or bal <= balance_gt:
150                 return False
151         return True
152
```

```
153     return list(filter(pred, data))
```

Die beiden Codeausschnitte zeigen die Umsetzung derselben Filterlogik in imperativer und funktionaler Form. In der imperativen Version (Listing 1) wird der Datensatz explizit mit einer Schleife durchlaufen, und die Filterbedingungen werden über Kontrollflussanweisungen wie `continue` realisiert. Zwischenergebnisse werden dabei in einer veränderlichen Liste gesammelt.

In der funktionalen Version (Listing 2) wird die Filterlogik in einer separaten Prädikatfunktion gekapselt und anschließend deklarativ mit der Higher-Order-Funktion `filter` auf den gesamten Datensatz angewendet. Dadurch entfällt expliziter Kontrollfluss, und die Logik beschreibt direkt, welche Elemente beibehalten werden sollen, anstatt wie iteriert werden muss.

6.1.2 Beispiel B: Gruppierung mit `reduce` (Group by)

Für mehrere Auswertungen (z. B. *Group by Education*) müssen Datensätze nach einem Feldwert gruppiert werden. In Version 1.0 wird dafür schrittweise ein Dictionary aufgebaut und jede Zeile wird in eine veränderliche Liste eingefügt. Version 2.0 formuliert dieselbe Operation als funktionale Faltung (`reduce`), wobei der Akkumulator als unveränderliche Struktur behandelt und bei jedem Schritt neu erzeugt wird.

Imperative Umsetzung (Version 1.0)

Listing 3: Imperative Gruppierung mit Schleifen und veränderlichem Zustand aus `imperative_version.py`

```
234
235 def group_by_key(data: List[Dict[str, Any]], key: str) -> Dict[str,
    List[Dict[str, Any]]]:
236
237     groups: Dict[str, List[Dict[str, Any]]] = {}
238     for row in data:
239         k = row.get(key)
240         k_str = "" if k is None else str(k)
241         if k_str not in groups:
242             groups[k_str] = []
243         groups[k_str].append(row)
244     return groups
```

Funktionale Umsetzung (Version 2.0)

Listing 4: Funktionale Gruppierung mit `reduce` aus `functional_version.py`

```
424
425 def group_by_key(data: List[Dict[str, Any]], key: str) -> Dict[str,
    Tuple[Dict[str, Any], ...]]:
426
427     def add(acc: Dict[str, Tuple[Dict[str, Any], ...]], row: Dict[str,
    Any]) -> Dict[str, Tuple[Dict[str, Any], ...]]:
```

```
428             k = row.get(key)
429             k_str = "" if k is None else str(k)
430             existing_group = acc.get(k_str, ())
431             new_group = existing_group + (row,)
432             return {**acc, k_str: new_group}
433 
434     return reduce(add, data, {})
```

Die folgenden Listings zeigen dieselbe Funktionalität zur Gruppierung von Datensätzen, einmal in imperativer (Listing 3) und einmal in funktionaler Umsetzung (Listing 4). Beide Ausschnitte stammen aus den jeweiligen Implementierungsdateien und wurden zur besseren Vergleichbarkeit isoliert dargestellt.

Die imperative Variante arbeitet mit veränderlichen Datenstrukturen (Dictionary und Listen) und verändert diese schrittweise. Die funktionale Variante beschreibt denselben Prozess als Abbildung von *Akkumulator + Element* auf einen neuen Akkumulator und vermeidet dabei mutierende Operationen. Dadurch wird die Gruppierungslogik deklarativer und lässt sich direkt für mehrere Auswertungen wiederverwenden (z. B. Bildung, Beruf oder Familienstand).

Darüber hinaus ist diese Gruppierungsfunktion kein isoliertes Demonstrationsbeispiel, sondern ein zentrales Bestandteil der Produktfunktionalität des Programms. Sie wird unter anderem in den Menüoptionen 5 (Group by education), 6 (Group by marital) sowie 7 (Vergleich zweier Gruppen) verwendet, um die Datensätze nach unterschiedlichen Kategorien zu gruppieren und darauf aufbauend aggregierte Kennzahlen zu berechnen.

6.1.3 Beispiel C: Transformation und Aggregation numerischer Werte (`map` und `sum`)

Die Berechnung des arithmetischen Mittels zeigt exemplarisch, wie numerische Transformation und Aggregation in der funktionalen Version ohne explizite Schleifen oder veränderliche Akkumulatoren umgesetzt werden können. Zunächst werden alle Werte mittels `map` in Gleitkommazahlen transformiert, anschließend erfolgt die Aggregation über `sum`.

Imperative Umsetzung (Version 1.0)

Listing 5: Imperative Berechnung des arithmetischen Mittels mit expliziter Schleife aus `imperative_version.py`

```
117 
118 def mean(values: List[float]) -> Optional[float]:
119 
120     if not values:
121         return None
122     s = 0.0
123     n = 0
124     for v in values:
125         s += float(v)
126         n += 1
```

```
127     if n == 0:
128         return None
129     return s / float(n)
```

Funktionale Umsetzung (Version 2.0)

Listing 6: Funktionale Berechnung des arithmetischen Mittels mit `map` und `sum` aus `functional_version.py`

```
294
295 def mean(values: List[float]) -> Optional[float]:
296
297     return (sum(map(float, values)) / float(len(values))) if values
    else None
```

Die beiden Listings zeigen die Berechnung des arithmetischen Mittels einmal in imperativer und einmal in funktionaler Form. In der imperativen Version (Listing 5) wird ein veränderlicher Akkumulator (`s`) sowie ein expliziter Zähler (`n`) verwendet, die innerhalb einer Schleife schrittweise aktualisiert werden.

In der funktionalen Version (Listing 6) wird dieselbe Berechnung deklarativ formuliert: Die Transformation der Werte erfolgt über `map`, die Aggregation über `sum`. Ein expliziter Schleifenindex oder ein veränderlicher Zwischenzustand ist nicht notwendig. Dadurch entspricht der Code direkt der mathematischen Definition des Mittelwerts und ist kompakter sowie leichter zu lesen.

6.1.4 Zusammenfassung der Unterschiede

- **Lesbarkeit:** Version 2.0 ist kompakter und beschreibt häufig *was* berechnet wird, statt *wie*.
- **Wartbarkeit:** Wiederverwendbare Hilfsfunktionen (Prädikate, Mapping-Funktionen) reduzieren Duplikation.
- **Fehleranfälligkeit:** Weniger mutable Zwischenzustände und weniger Hilfsvariablen verringern typische Fehlerquellen.

7 Fazit

7.1 Nutzen funktionaler Elemente

Der Einsatz funktionaler Sprachelemente hat in diesem Projekt mehrere Vorteile gebracht. Insbesondere Filter- und Transformationsoperationen konnten kompakter und klarer formuliert werden.

7.2 Vereinfachung durch Refactoring

Im Vergleich zur imperativen Version ist die funktionale Umsetzung nicht zwingend kürzer, jedoch stärker strukturiert. Verschachtelte Schleifen und mutable Hilfsvariablen werden häufig

durch deklarative Ausdrücke ersetzt, welche Transformation, Selektion und Aggregation klar voneinander trennen. Der Fokus liegt somit weniger auf der reinen Code-Länge, sondern auf Lesbarkeit, Wartbarkeit und einer geringeren Fehleranfälligkeit.

7.3 Kritische Evaluation: Grenzen und Trade-offs der funktionalen Umsetzung

Die funktionale Version bringt konzeptionelle Vorteile in Bezug auf Strukturierung und Ausdrucksstärke, geht jedoch auch mit messbaren Nachteilen einher, die im Projekt klar sichtbar wurden.

Performance-Trade-offs Die Funktion `group_by_key` verwendet in der funktionalen Implementierung ausschließlich immutable Datenstrukturen. Bei jedem Reduktionsschritt wird ein neues Dictionary mittels `{**acc, key: value}` erzeugt. Dadurch müssen alle bereits gesammelten Einträge kopiert werden.

Dies führt zu einer Zeitkomplexität von $\mathcal{O}(n^2)$, während die imperative Version durch direkte Mutation des Dictionaries eine $\mathcal{O}(n)$-Komplexität erreicht. Bei einem Datensatz mit 7474 Einträgen ist dieser Unterschied zwar noch praktisch handhabbar, skaliert jedoch schlecht für größere Datenmengen.

Die funktionale Variante ist somit primär didaktisch sinnvoll, stellt aber keine performante Produktionslösung dar.

Quantitativer Vergleich der Implementierungen Zur objektiven Bewertung wurden beide Versionen hinsichtlich einiger Metriken verglichen:

Metrik	**Imperativ**	**Funktional**
LOC (ohne Docstrings/Kommentare/Leerzeilen)	434	396
Durchschnittliche Funktionslänge (LOC)	21.15	11.89
Maximale Verschachtelungstiefe (AST-Blöcke)	13	6

Die Kennzahlen wurden automatisiert aus dem Quelltext bestimmt (`tests/metrics.py`). Dabei werden physische Codezeilen ohne Leerzeilen, Kommentare und Docstrings gezählt; Funktionslängen basieren auf `lineno/end_lineno` des Python-AST.

Die Messwerte zeigen, dass die funktionale Version in diesem Projekt *nicht länger*, sondern sogar etwas *kürzer* ausfällt (396 vs. 434 LOC). Der deutlichere Unterschied liegt jedoch in der **Struktur**:

- Die imperative Version weist eine deutlich höhere maximale Verschachtelungstiefe auf (13 vs. 6). Dies entsteht vor allem durch die stärker kontrollflussgetriebene Umsetzung (mehrstufige Schleifen-/If-Kombinationen und Menülogik).
- Die durchschnittliche Funktionslänge ist in der imperativen Version fast doppelt so hoch (21.15 vs. 11.89 LOC), was auf größere, monolithischere Funktionsblöcke hindeutet.

Die funktionale Version verteilt Logik häufiger auf kleinere Funktionen und nutzt Komposition (z. B. `pipe`/`compose`) sowie deklarative Operationen (`map`/`filter`/`reduce`). Dadurch sinken im Mittel Funktionslänge und Verschachtelung, auch wenn einzelne funktionale Konstrukte für Ungeübte weniger intuitiv sein können.

Kognitive Komplexität Ein Nachteil ist die höhere Einstiegshürde: Konstrukte wie `reduce`, Funktionenkombinationen (`pipe`, `compose`) und verschachtelte Lambdas sind für viele Entwickelnde weniger direkt nachvollziehbar als klassische Schleifen. In Teams ohne funktionale Erfahrung kann die imperative Version daher zunächst wartungsfreundlicher wirken, obwohl sie strukturell stärker verschachtelt ist.

Fazit der kritischen Betrachtung Die funktionale Umsetzung bietet in diesem Projekt messbare Vorteile in Bezug auf geringere Verschachtelung und kürzere Funktionen, kann jedoch zu folgenden Trade-offs führen:

- höherer Abstraktionsgrad und damit höhere Einstiegshürde,
- potenziell schlechtere Laufzeit bei strikt immutablen Mustern (z. B. Dictionary-Kopien in `group_by_key`),
- Debugging kann bei stark verschachtelter Komposition schwieriger sein als bei linearem Kontrollfluss.

Die funktionale Version ist damit im Projekt primär als didaktisches und architektonisches Vergleichsmodell zu verstehen; für Produktionscode wären je nach Kontext hybride Lösungen (funktionale Pipelines bei gleichzeitiger pragmatischer Mutation in Hotspots) naheliegend.

7.4 Bewusste Designentscheidungen

Im Rahmen der Transformation numerischer Variablen wurden bewusst unterschiedliche mathematische Abbildungen eingesetzt. Neben einer fachlich realistischen Transformation mittels Logarithmus log(balance) wurde zusätzlich eine einfache quadratische Transformation $\text{balance}^2 + 1$ verwendet.

Der Logarithmus ist insbesondere im wirtschaftlichen Kontext sinnvoll, da er rechtsschiefe Verteilungen reduziert und Ausreißer abschwächt. Allerdings ist diese Transformation nur für positive Werte definiert und erfordert zusätzliche Prüfungen des Wertebereichs.

Die quadratische Transformation hingegen ist für alle numerischen Werte definiert und eignet sich besonders gut, um funktionale Abbildungen unabhängig von Sonderfällen zu demonstrieren. Dadurch konnte der Fokus im Projekt gezielt auf den Vergleich funktionaler Programmierkonzepte gelegt werden, ohne die Darstellung durch zusätzliche Randfallbehandlung zu verkomplizieren.

7.5 Wiederverwendung funktionaler Konzepte

Die im Projekt eingesetzten funktionalen Konzepte eignen sich auch für zukünftige Datenanalyse- und Auswertungsprojekte und würden insbesondere in folgenden Bereichen erneut verwendet werden:

- **Datenfilterung** über Prädikatfunktionen und deren Kombination (z. B. mittels `filter()`, `combine_predicates`).
- **Gruppierung von Datensätzen** durch funktionale Faltungen (z. B. `group_by_key` mit `reduce`).
- **Aggregation und Kennzahlenberechnung**, etwa für Mittelwerte, Varianzen oder Erfolgsquoten (`mean`, ANOVA, Zähloperationen).
- **Transformation von Datenpipelines** durch funktionale Verkettung von Verarbeitungsschritten (`pipe`, `compose`).
- **Kapselung von Logik in Higher-Order Functions**, z. B. durch Funktionen, die andere Funktionen erzeugen (`create_balance_filter`).
- **Nebenwirkungsfreie Datenverarbeitung** durch den Einsatz unveränderlicher Datenstrukturen (Tupel, neue Dictionaries statt Mutation).
- **Testbarkeit und Vergleichbarkeit**, da reine Funktionen leicht automatisiert getestet und zwischen Implementierungen verglichen werden können.

7.6 Grenzen funktionaler Elemente in Python

Obwohl Python zahlreiche funktionale Sprachelemente bereitstellt, handelt es sich nicht um eine rein funktionale Programmiersprache. Im Vergleich zu funktionalen Sprachen wie Haskell werden zentrale Konzepte wie Immutabilität oder Nebenwirkungsfreiheit nicht erzwungen, sondern lediglich konventionell eingehalten.

Variablen können weiterhin verändert werden, und Funktionen können unbeabsichtigt Seiteneffekte erzeugen. Die korrekte Anwendung funktionaler Konzepte hängt somit stark von der Disziplin der Entwickelnden ab und wird nicht durch den Sprachkern abgesichert.

Diese Einschränkungen wurden im Projekt bewusst in Kauf genommen, da der Fokus auf dem Vergleich imperativer und funktionaler Denkweisen innerhalb einer praxisnahen Sprache lag.

In einer rein funktionalen Sprache könnten diese Konzepte konsistenter und durch den Compiler abgesichert umgesetzt werden. Python eignet sich jedoch besonders gut, um funktionale Programmierkonzepte schrittweise in bestehende imperative Arbeitsweisen zu integrieren.

7.7 Anwendungsfälle im beruflichen Umfeld

Im betrieblichen Kontext eignen sich funktionale Sprachelemente besonders für:

- Datenanalyse
- Reporting
- Verarbeitung von Log- oder Kundendaten
- Automatisierte Auswertungen

Zusammenfassend lässt sich sagen, dass funktionale Programmieransätze eine sinnvolle Ergänzung zur imperativen Programmierung darstellen und in vielen Anwendungsfällen zu einer klareren und effizienteren Lösung führen.

8 Ausführung des Programms

Der vollständige Quellcode befindet sich im Projekt-Repository (vgl. [4]). Die Analysen wurden im Rahmen dieses Projekts primär in Jupyter Notebooks durchgeführt, um Auswertungen und Ergebnisse nachvollziehbar zu dokumentieren.

Alternativ kann das Programm auch als Python-Skript ausgeführt werden. Die notwendigen Schritte zur Ausführung sind im Repository dokumentiert.

9 Hinweis zur Entstehung der Arbeit

Zur Unterstützung bei der Dokumentation und beim Refactoring wurden KI-basierte Werkzeuge konsultiert. Die Projektarbeit wurde eigenständig umgesetzt; sämtliche Ergebnisse, Implementierungen und Auswertungen wurden von den Autoren geprüft, verstanden und verantwortet.

10 Quellen

Literatur

[1] S. Moro, R. Laureano, P. Cortez. *Using Data Mining for Bank Direct Marketing: An Application of the CRISP-DM Methodology.* Proceedings of the European Simulation and Modelling Conference (ESM 2011).

[2] Maringer, Dietmar und Kitzing, Emil. *Multivariate Datenanalyse – Assignment 1.* WWZ, Universität Basel, Herbstsemester 2024.

[3] Montgomery, Douglas C. *Design and Analysis of Experiments.* 9th Edition, Wiley, 2022.

[4] Projekt-Repository: https://github.com/stoicfist/Modul-323-Projektarbeit

Hinweis: Der vollständige Quellcode wurde in dieser Dokumentation bewusst nicht vollständig abgedruckt, um den Fokus auf Konzept und Vergleich zu legen (vgl. [4]).

11 Appendix

11.1 Projekt-Repository

Der vollständige Quellcode des Projekts ist im öffentlichen GitHub-Repository verfügbar (vgl. [4]).

11.2 Weitere Beispielausgaben

11.2.1 Menüoption 7: Vergleich zweier Gruppen

```
Verfügbare Gruppen in 'education': primary, secondary, tertiary

======================================================================
                        VERGLEICH ZWEIER GRUPPEN
======================================================================
education | count | avg(age) | avg(balance) | avg(duration) | success
----------+-------+----------+--------------+---------------+--------
tertiary  |  2657 |     39.7 |      2404.61 |         382.4 |   62.5%
secondary |  3815 |     40.7 |      1709.54 |         387.2 |   52.2%

 Erfolgsquote (A-B): +10.3%
```

11.2.2 Menüoption 8: ANOVA-Auswertung

```
======================================================================
                    ANOVA-ÄHNLICHER F-WERT (BALANCE)
======================================================================
F(2, 7471) = 41.928

Interpretation: Deutliche Unterschiede der Mittelwerte zwischen Gruppen möglich
(hoher F-Wert).
```

11.2.3 Menüoption 1: Erfolgsquote gesamt

```
======================================================================
                              ERFOLGSQUOTE
======================================================================
Metric |  Value
-------+-------
Total  |   7474
Yes    |   4137
Quote  |  55.4%
```

11.3 Interaktive Konsolen-Ausführung (CLI)

Neben der Ausführung im Jupyter Notebook kann das Programm auch direkt als Python-Skript gestartet werden. Dabei wird eine textbasierte Benutzeroberfläche angezeigt, welche die Bedienung über ein Menü ermöglicht:

```
======================================================================
                        BANK MARKETING - CLI
======================================================================
```

```
                        Datensätze geladen: 7474
======================================================================
                                MENU
======================================================================
1) Erfolgsquote gesamt
2) Filter setzen (housing/loan/balance>X)
3) Transformationen (log(balance), balance^2+1)
4) duration Analyse + optional Buckets
5) Group by education
6) Group by marital
7) Vergleich zweier Gruppen
8) ANOVA-ähnlicher F-Wert (balance)
q) Quit
Auswahl: 7
======================================================================
                        VERGLEICH ZWEIER GRUPPEN
======================================================================
Gruppierungsfeld (education/marital/job) [Default: education]:
Verfügbare Gruppen:
primary, secondary, tertiary
Gruppe A: tertiary
Gruppe B: secondary
education | count | avg(age) | avg(balance) | avg(duration) | success
----------+-------+----------+--------------+---------------+--------
tertiary  |  2657 |     39.7 |      2404.61 |         382.4 |   62.5%
secondary |  3815 |     40.7 |      1709.54 |         387.2 |   52.2%
 Erfolgsquote (A-B): +10.3%
```