Spis treści

Wstęp

Pierwsze wydanie książki *Uczenie maszynowe w praktyce* napisałem sfrustrowany brakiem dyscypliny swoich współpracowników. W roku 2009 pracowałem nad wieloma projektami związanymi z uczeniem maszynowym i zauważyłem, że gdy tylko wprowadzaliśmy pojęcie maszyn wektorów nośnych, sieci neuronowych itp., nagle podstawowe praktyki kodowania po prostu znikały.

Moją odpowiedzią była książka *Uczenie maszynowe w praktyce*. W tym czasie pisałem 100% swojego kodu w języku Ruby i napisałem tę książkę z wykorzystaniem tego języka. Cieszę się, że mogę przedstawić nowe wydanie tej książki, przepisane z wykorzystaniem języka Python, co stanowiło nie lada wyzwanie. Przejrzałem większość rozdziałów, pozmieniałem przykłady i zaktualizowałem treść, aby była bardziej przydatna dla osób, które będą pisać kod związany z uczeniem maszynowym. Mam nadzieję, że spodoba się to czytelnikom.

Jak wspominałem w pierwszym wydaniu, moje drzwi zawsze stoją otworem. Jeśli ktoś chciałby ze mną porozmawiać, może się ze mną skontaktować pod adresem *matt@matthewkirk.com*. A jeśli ktoś trafi kiedyś do Seattle, możemy umówić się na spotkanie przy kawie.

Konwencje wykorzystywane w tej książce

Następujące konwencje typograficzne są używane w tej książce:

Kursywa
: Wskazuje nowe pojęcia, adresy URL, adresy e-mail, nazwy plików i rozszerzenia plików.

`Czcionka stałopozycyjna`
: Używana w przykładach programów, a także w treści akapitów przy odwołaniach do elementów programów, takich jak nazwy zmiennych lub funkcji, bazy danych, typy danych, zmienne środowiskowe, instrukcje i słowa kluczowe.

`Czcionka stałopozycyjna z pogrubieniem`
: Pokazuje polecenia lub inny tekst, który powinien być dosłownie wpisany przez użytkownika.

`Czcionka stałopozycyjna z kursywą`
: Pokazuje tekst, który powinien być zastąpiony wartościami podanymi przez użytkownika lub wynikającymi z kontekstu.

Ten element symbolizuje uwagę ogólną.

Korzystanie z przykładów kodu

Materiały dodatkowe (przykłady kodu, ćwiczenia, itd.) są dostępne do pobrania pod adresem *http://github.com/thoughtfulml/examples-in-python*.

Ta książka ma pomóc czytelnikom w realizacji własnych projektów. Jeśli jakiś kod przykładowy jest oferowany w tej książce, można z niego korzystać w swoich programach i dokumentacji. Nie trzeba uzyskiwać od nas pozwolenia, o ile nie kopiuje się znacznej ilości kodu. Na przykład napisanie programu, który wykorzystuje kilka fragmentów kodu z tej książki, nie wymaga zezwolenia. Sprzedawanie lub dystrybucja dysku CD-ROM z przykładami kodu z książek wydawnictwa O'Reilly wymaga pozwolenia. Cytowanie tej książki i przykładów kodu w odpowiedzi na czyjeś pytanie nie wymaga pozwolenia. Włączenie znaczącej ilości kodu przykładowego z tej książki do dokumentacji swojego produktu wymaga pozwolenia.

Doceniamy powołanie się na źródło, ale tego nie wymagamy. Zwykle zawiera ono tytuł, autora, wydawcę i numer ISBN książki.

Jeśli ktoś ma wątpliwości, czy użycie przykładów kodu z tej książki wymaga dodatkowego zezwolenia, może się z nami skontaktować pod adresem *permissions@oreilly.com*.

Jak się z nami skontaktować

Komentarze i pytania dotyczące tej książki należy kierować na adres wydawcy:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (w USA lub Kanadzie)
707-829-0515 (połączenie międzynarodowe lub lokalne)
707-829-0104 (fax)

Przygotowaliśmy stronę WWW dla tej książki, na której umieszczamy erratę, przykłady i wszelkie dodatkowe informacje. Strona ta jest dostępna pod adresem *http://bit.ly/thoughtful-machine-learning-with-python*.

Aby przesłać komentarz lub zadać pytanie techniczne dotyczące tej książki, należy wysłać wiadomość pod adres *bookquestions@oreilly.com*.

Więcej informacji na temat naszych książek, kursów i konferencji można znaleźć w naszej witrynie WWW pod adresem *http://www.oreilly.com*.

Można nas znaleźć na Facebooku pod adresem: *http://facebook.com/oreilly*.

Można nas śledzić w serwisie Twitter: *http://twitter.com/oreillymedia*.

Można nas oglądać na YouTube: *http://www.youtube.com/oreillymedia*.

Podziękowania

Musiałem poczekać ponad rok na ukończenie tej książki. Zdiagnozowany u mnie rak jądra i nagła śmierć mojego ojca zmusiły mnie do przerwy i refleksji, zanim mogłem ponownie zabrać się za pisanie. Choć zajęło to dłużej niż wcześniej szacowałem, jestem całkiem zadowolony z wyniku.

Jestem wdzięczny wszystkim w wydawnictwie O'Reilly za wsparcie, jakie otrzymałem podczas pisania tej książki. Chciałbym szczególnie wyróżnić swoją redaktorkę Shannon Cut, która była dla mnie podporą i wsparciem; Liz Rush, która jako recenzent techniczny wytrwała przez cały proces pisania tej książki; Stephena Elstona, który dał mi wiele pomocnych uwag; Mike'a Loukidesa, za wsparcie mojego pomysłu i pomoc w przekształceniu go w dwie wydane książki; Alexeya Porotnikova, który pomógł mi opracować przykłady kodu w języku Python.

Chciałbym też specjalnie podziękować Alexeyowi Porotnikovowi (*https://github.com/alpo*) za pracowitą pomoc w przekonwertowaniu wszystkich tych przykładów z języka Ruby do Python, a także z wersji Python 2 do Python 3. Naprawdę dziękuję!

Jestem wdzięczny swoim przyjaciołom, szczególnie Curtis Fanta. Znamy się od małego. Dziękuję za poświęcony mi czas.

Chciałbym też podziękować swojej rodzinie. Moim bratankom Zoe i Darby za ciekawość i zdumienie. Bratu Jake'owi za zabawianie mnie nową muzyką i filmami. Mamie Carol za to, że pozwalała mi odkrywać świat i radziła zająć się fizyką (czego nie posłuchałem). Wszyscy wiele dla mnie znaczycie.

Rodzinie Le za traktowanie mnie jak swojego. Dziękuję Lilianie za spotkania przy klockach Lego oraz Sayone i Alyssie za rozpromienianie mojego życia. Martinowi i Hanowi za ciągłe wsparcie i miłość. Thanh (tacie) i Kim (mamie) za karmienie mnie ponad miarę oraz za podarowanie mi mierników i książek na temat wzmacniaczy operacyjnych. Dziękuję za bycie częścią mojego życia.

Babci, która ciągle pytała, kiedy zobaczy okładkę. Zawsze pobudzała mnie do działania, czy to w harcerstwie, czy we własnym biznesie. Dziękuję za bycie zawsze przy mnie.

Dziękuję Sophii, mojej żonie. Rok temu byliśmy w szpitalnej sali, gdzie wpompowano we mnie mnóstwo środków przeciwbólowych…i przeżyliśmy. Jesteś stałą podporą mojego dorosłego życia. Gdy podejmuję się jakiegoś śmiałego wyzwania (jak napisanie książki), zawsze odkładasz swoje potrzeby na bok, aby się mną zająć. Jesteś dla mnie całym światem.

Wreszcie dziękuję mojemu tacie. Brakuje mi Twoich odwiedzin i naszych wypraw pod namiot do lasu. Chciałbym dzielić się z Tobą tym osiągnięciem, ale doceniam czas, jaki spędziliśmy razem. Ta książka jest dla Ciebie.

Rozdział 1

W przybliżeniu prawdopodobnie poprawne oprogramowanie

Lecąc samolotem, korzystamy z jednej z najbezpieczniejszych form podróżowania na świecie. Szanse na utratę życia podczas lotu samolotem wynoszą 1 na 29,4 milionów[1], co oznacza, że decydując się na zawód pilota pasażerskiego, można nigdy nie uczestniczyć w wypadku podczas całej 40-letniej kariery. Te szanse są szokujące, biorąc pod uwagę, jak skomplikowanymi urządzeniami są samoloty. Nie zawsze jednak tak było.

Rok 2014 był zły dla lotnictwa; odnotowano wtedy 824 zgony związane z lotnictwem[2], w tym zaginięcie samolotu linii Malaysia Air. W roku 1929 było 257 ofiar[3] wypadków lotniczych. Wygląda to, jakby bezpieczeństwo lotów uległo pogorszeniu, dopóki nie zdamy sobie sprawy, że w samych Stanach Zjednoczonych odbywa się obecnie ponad 10 milionów lotów rocznie, natomiast w roku 1929 było ich znacznie mniej – mniej więcej od 50000 do 100000. Oznacza to, że ogólne prawdopodobieństwo śmierci w katastrofie lotniczej zmalało od 1929 do 2014 roku z 0,25% do 0,00824%.

Podróże lotnicze zmieniły się przez lata, tak samo jak tworzenie oprogramowania. Podczas gdy w roku 1929 tworzenie oprogramowania w postaci znanej obecnie nie istniało, to przez ostatnie lata zbudowaliśmy i zawaliliśmy wiele projektów programistycznych.

Do niedawnych przykładów należą takie projekty programistyczne, jak uruchomienie amerykańskiego serwisu healthcare.gov, który okazał się finansową katastrofą, kosztującą około 634 miliony dolarów[4]. Jeszcze gorsze są projekty programistyczne, które mają katastrofalne błędy. W roku 2013 giełda NASDAQ została zamknięta z powodu błędu oprogramowania i została ukarana grzywną w wysokości 10 milionów USD[5]. W roku 2014 byliśmy świadkami rozprzestrzeniania się błędu Heartbleed, który naraził na niebezpieczeństwo

1 *http://www.statisticbrain.com/airplane-crash-statistics/*
2 *http://bit.ly/2014-aviation*
3 *http://bit.ly/casualties-1929*
4 *http://bit.ly/cost-healthcaregov*
5 *http://reut.rs/2i2HfgS*

wiele witryn korzystających z SSL. W konsekwencji firma CloudFlare musiała odwołać ponad 100000 certyfikatów SSL, co kosztowało ich wiele milionów[6].

Oprogramowanie i samoloty mają jedną wspólną cechę: są skomplikowane i skutki ewentualnej awarii są katastrofalne i widoczne publicznie. Linie lotnicze były w stanie zapewnić bezpieczeństwo podróży i zmniejszyć prawdopodobieństwo katastrof lotniczych o ponad 96%. Niestety nie możemy powiedzieć tego samego o oprogramowaniu, które staje się jeszcze bardziej skomplikowane. Katastrofalne błędy uderzają w nas regularnie, powodując miliardowe straty.

Dlaczego podróże lotnicze stały się tak bezpieczne, a oprogramowanie tak wadliwe?

Prawidłowe pisanie oprogramowania

Między rokiem 1929 a 2014 samoloty stały się bardziej skomplikowane, większe i szybsze. Jednak z tym wzrostem wiązały się też większe regulacje ze strony FAA oraz instytucji międzynarodowych, a także kultura stosowania list kontrolnych przez pilotów.

Choć technologie i sprzęt komputerowy zmieniały się bardzo szybko, to uruchamiane na nich oprogramowanie zmieniało się znacznie wolniej. Nadal korzystamy głównie z kodu proceduralnego i zorientowanego obiektowo, który nie wykorzystuje w pełni możliwości obliczeń równoległych. Programiści podjęli jednak kroki w kierunku opracowania wytycznych dotyczących pisania oprogramowania oraz stworzenia kultury testowania kodu. Doprowadziło to do przyjęcia zasad SOLID i metodologii TDD. SOLID jest zestawem zasad, które prowadzą nas do pisania lepszego kodu, a TDD oznacza projektowanie sterowane testami (test-driven design) lub programowanie sterowane testami (test-driven development). Omówimy te dwa modele i ich związki z prawidłowym pisaniem oprogramowania oraz porozmawiamy o refaktoringu oprogramowania.

SOLID

Zasady SOLID stanowią platformę, która pomaga w projektowaniu lepszego kodu zorientowanego obiektowo. W taki sam sposób, jak FAA (Federal Aviation Administration – Federalna Administracja Lotnictwa) definiuje, jak *powinny* działać linie lotnicze lub samoloty, tak samo zasady SOLID mówią nam, jak *powinniśmy* tworzyć oprogramowanie. Naruszenia regulacji FAA czasem się zdarzają i mogą mieć różne skutki, od katastrofalnych do drobnych. To samo odnosi się do zasad SOLID. Zasady te mogą mieć czasem ogromne znaczenie, ale najczęściej są po prostu wskazówkami. Zasady SOLID wprowadził Robert Martin jako pięć zasad tworzenia oprogramowania. Motywem do ich opracowania było pisanie lepszego kodu, który będzie łatwy w utrzymaniu, zrozumiały i stabilny. Michael Feathers wymyślił mnemotechniczny skrót *SOLID* (od pierwszych liter nazw poszczególnych zasad) do łatwiejszego ich zapamiętania.

6 *http://bit.ly/cost-heartbleed*

SOLID oznacza:

- SRP – Single Responsibility Principle (zasada pojedynczej odpowiedzialności)
- OCP – Open/Closed Principle (zasada otwarte-zamknięte)
- LSP – Liskov Substitution Principle (zasada podstawienia Liskov)
- ISP – Interface Segregation Principle (zasada rozdzielenia interfejsów)
- DIP – Dependency Inversion Principle (zasada odwrócenia zależności)

Zasada pojedynczej odpowiedzialności

Zasada pojedynczej odpowiedzialności (SRP) stała się jednym z bardziej rozpowszechnionych elementów pisania dobrego kodu zorientowanego obiektowo. Pojedyncza odpowiedzialność powoduje definiowanie prostych klas lub obiektów. Tę samą mentalność można stosować w programowaniu funkcjonalnym, tworząc prostsze funkcje. Cała koncepcja opiera się na prostocie. Dany element oprogramowania powinien zajmować się jedną i tylko jedną kwestią. Dobrym przykładem naruszenia zasady SRP jest scyzoryk wielofunkcyjny (rysunek 1-1). Może zrobić niemal wszystko, ale użyteczność poszczególnych elementów jest ograniczona.

Rysunek 1-1 *Scyzoryk wielofunkcyjny ma zbyt wiele odpowiedzialności.*

Zasada otwarte-zamknięte

Zasada OCP, określana też czasem mianem hermetyzacji, mówi, że obiekty powinny być otwarte na rozszerzanie, ale nie na modyfikowanie. Można to pokazać na przykładzie obiektu zliczającego, z którym jest związany wewnętrzny licznik. Obiekt ten ma metody `increment` i `decrement`. Obiekt ten nie powinien zezwalać innym elementom na zmienianie wewnętrznego licznika poza zdefiniowanym interfejsem API, ale może być rozszerzany (np., aby powiadamiać kogoś o zmianie licznika przez obiekt typu `Notifier`).

Zasada podstawienia Liskov

Zasada LSP określa, że dowolny podtyp powinien być łatwo zastępowalny elementem występującym niżej w drzewie obiektów, bez efektów ubocznych. Na przykład model samochodu można podstawić zamiast rzeczywistego samochodu.

Zasada rozdzielenia interfejsów

Zasada ISP stwierdza, że występowanie wielu interfejsów specyficznych dla klientów jest lepsze niż ogólny interfejs dla wszystkich klientów. Ta zasada dotyczy upraszczania wymiany danych pomiędzy różnymi elementami. Dobrym przykładem jest segregowanie śmieci na organiczne, do ponownego przetworzenia i pozostałe odpady. Zamiast jednego dużego kubła, mamy trzy dla poszczególnych typów śmieci.

Zasada odwrócenia zależności

Zasada DIP radzi nam polegać na abstrakcjach, a nie na elementach konkretnych. Mówi nam, że powinniśmy budować drzewo dziedziczenia obiektów. Przykład użyty w oryginalnej pracy[7] Roberta Martina wyjaśnia, że powinniśmy utworzyć klasę `KeyboardReader` dziedziczącą po ogólnej klasie `Reader`, zamiast umieszczać wszystko w jednej klasie. Pasuje to też dobrze do rady Arthura Riela dotyczącej unikania *boskich klas*, pochodzącej z jego pracy *Heurystyka projektowania zorientowanego obiektowo*. Choć moglibyśmy bezpośrednio przylutować przewód prowadzący z gitary do wzmacniacza, byłoby to pewnie niewydolne i nie brzmiałoby zbyt dobrze.

Zestaw zasad SOLID przeszedł próbę czasu i został przedstawiony w wielu książkach autorstwa Martina i Feathersa, a także pojawił się w książce Sandi Metz *Praktyczne projektowanie zorientowane obiektowo w języku Ruby*[8]. Zestaw ten ma służyć jako wskazówki i przypominać nam o prostych sprawach przy programowaniu, aby tworzony przez nas kod był jak najlepszy. Wskazówki te pomagają pisać poprawne architektonicznie oprogramowanie.

Testowanie albo TDD

We wczesnych dniach lotnictwa piloci nie używali list kontrolnych do sprawdzania, czy samolot jest gotowy do startu. W książce *The Right Stuff* Tom Wolfe pisał, że większość wczesnych pilotów testowych takich jak Chuck Yeager kierowało się wyczuciem i swoim

7 Robert Martin, „The Dependency Inversion Principle", *http://bit.ly/the-DIP*.
8 *http://poodr.info/*

talentem przy opanowywaniu złożoności samolotu. To prowadziło również do tego, że jedna czwarta pilotów testowych zginęła w akcji.[9]

Obecnie sprawy mają się inaczej. Przed startem piloci przeprowadzają mnóstwo weryfikacji. Niektóre z nich mogą wydawać się niepotrzebnie uciążliwe, jak na przykład przedstawienie się imieniem innym członkom załogi. Wyobraźmy sobie jednak, że samolot wpadnie w korkociąg i trzeba natychmiast poinformować kogoś o problemie. Nie znając imienia, ciężko byłoby się komunikować.

To samo można odnieść wobec dobrego oprogramowania. Regularne uruchamianie zestawu systematycznych sprawdzeń w celu przetestowania, czy nasze oprogramowanie działa poprawnie, czy nie, pozwala na jego spójne działanie.

We wczesnych dniach tworzenia oprogramowania większość testów była przeprowadzana po napisaniu właściwego oprogramowania (jest to tak zwany model kaskadowy[10], wykorzystywany przez NASA i inne organizacje do projektowania oprogramowania i testowania go). Współdziałało to dobrze z powszechnym wówczas stylem zarządzania projektami. Podobnie do sposobu budowania samolotów, w przypadku oprogramowania najpierw powstawał jego projekt, następnie pisano kod zgodnie ze specyfikacjami, a potem testowano program przed dostarczeniem go klientowi. Ta metoda testowania mogła zajmować miesiące a nawet lata. Doprowadziło to do sformułowania *Manifestu programowania zwinnego*[11], a także do powstania kultury testowania i programowania sterowanego testami (TDD), zainicjowanego przez Kenta Becka, Warda Cunninghama i wielu innych.

Koncepcja programowania sterowanego testami jest prosta: najpierw piszemy test sprawdzający, co chcemy osiągnąć; wykonujemy go, potwierdzając, że daje wynik negatywny; piszemy kod naprawiający test; a następnie poprawiamy kod, aby dopasować go do wskazówek SOLID. Choć wiele osób zwraca uwagę, że wydłuża to cykl tworzenia oprogramowania, to znacząco ogranicza liczbę błędów w kodzie i poprawia stabilność działania oprogramowania.[12]

Samoloty, które mają niską tolerancję na błędy, często działają w ten sam sposób. Zanim pilot będzie mógł zasiąść za sterami Boeinga 787, spędza X godzin w symulatorze lotów, żeby zrozumieć i sprawdzić swoją znajomość samolotu. Samoloty są testowane przed startem, a później również w trakcie lotu. Z nowoczesnym tworzeniem oprogramowania jest bardzo podobnie. Testujemy swoją wiedzę, pisząc testy przed wdrożeniem oprogramowania, a także monitorując już wdrożone oprogramowanie.

Nadal pozostaje jednak jeden problem: w rzeczywistości, w której nic nie pozostaje takie samo, napisanie testu nie sprawia, że kod będzie dobry. David Heinemer Hanson w swojej popularnej prezentacji na temat szkód wynikających z podejścia sterowanego

9 Atul Gawande, *The Checklist Manifesto* (New York: Metropolitan Books), str. 161.

10 *https://en.wikipedia.org/wiki/Waterfall_model*

11 *http://agilemanifesto.org/*

12 Nachiappan Nagappan i in., „Realizing Quality Improvement through Test Driven Development: Results and Experience of Four Industrial Teams", *Empirical Software Engineering* 13, no. 3 (2008): 289 – 302, *http://bit.ly/Nagappanetal.*

testami[13] pokazał, jak ślepe oddanie technikom TDD i SOLID może prowadzić do skomplikowanego kodu. Większość jego uwag jest związanych z niepotrzebnymi komplikacjami wynikającymi z wyciągania drobnych fragmentów kodu do osobnych klas lub pisania kodu, który łatwo testować, ale ciężko czytać. Tutaj pojawia się kolejny czynnik związany z prawidłowym pisaniem oprogramowania: refaktoring.

Refaktoring

Refaktoring jest jedną z najtrudniejszych do wytłumaczenia osobom nieprogramującym praktyk programistycznych. Podczas lotu samolotem widzimy jedynie 20% z tego, co sprawia, że samolot leci. Pod warstwami aluminium i tytanu znajdują się złożone systemy elektryczne, które zasilają oświetlenie awaryjne, hydraulikę, konstrukcję nośną, która ma być lekka, a przy tym wytrzymała – zbyt wiele, aby wszystko tutaj wymienić. Wyjaśnianie komuś, co składa się na konstrukcję samolotu, jest jak wyjaśnianie komuś, że pod zlewem są rury.

Refaktoring bazuje na istniejącej strukturze i ją ulepsza. Polega na czyszczeniu istniejącego kodu, aby przeglądając go, łatwo było zrozumieć, co się dzieje. Samoloty są projektowane bardzo rygorystycznie, ale oprogramowanie nie. Oprogramowanie zmienia się bardzo szybko. Wiele firm nieustannie wdraża oprogramowanie w środowisku produkcyjnym. Całe to projektowanie nowych funkcji może czasem powodować tzw. „dług techniczny".

Dług techniczny, nazywany też *długiem projektowym* lub *długiem kodu*, jest metaforą dla słabego zaprojektowania systemu, który ulega degradacji w czasie życia projektów programistycznych. Problemem długu technicznego jest nawarstwianie się „odsetek", które w końcu blokują przyszły rozwój oprogramowania.

Jeśli ktoś pracował długo nad jakimś projektem, to zna to uczucie, gdy mamy szybkie nowe wersje w początkowym okresie rozwoju i stagnację pod koniec. Dług techniczny w wielu przypadkach wynika z braku testów albo niestosowania zasad SOLID.

Dług techniczny niekoniecznie jest zły – czasami trzeba projekty przepchnąć wcześniej, aby wspomóc rozwój biznesu – ale „niespłacanie" go sprawi, że narosną tak duże „odsetki", że projekt zostanie zniszczony. Sposobem na poradzenie sobie z tym problemem jest refaktoring kodu.

Poprzez refaktoring przybliżamy nasz kod do wskazówek SOLID i bazy kodu opartej na TDD. Stosując się do poniższych wskazówek, możemy czyścić istniejący kod i ułatwiać nowym programistom dołączanie do pracy nad jego dalszym rozwojem:

1. Stosowanie wskazówek SOLID
   a. Zasada pojedynczej odpowiedzialności
   b. Zasada otwarte-zamknięte
   c. Zasada podstawienia Liskov

13 *http://bit.ly/test-induced-damage*

d. Zasada rozdzielenia interfejsów
e. Zasada odwrócenia zależności

2. Implementowanie TDD (programowania/projektowania sterowanego testami)
3. Refaktoring kodu w celu uniknięcia narastania długu technicznego

Co tak właściwie sprawia, że oprogramowanie jest prawidłowe?

Pisanie prawidłowego oprogramowania

Pisanie prawidłowego oprogramowania jest dużo trudniejsze niż prawidłowe pisanie oprogramowania. W swojej książce *Specification by Example* Gojko Adzic twierdzi, że najlepszym podejściem do pisania oprogramowania jest utworzenie najpierw specyfikacji, a następnie bezpośrednia praca z użytkownikami tego oprogramowania. Dopiero po ukończeniu specyfikacji można pisać kod odpowiadający tej specyfikacji. W praktyce nie zawsze się to sprawdza – czasami świat nie jest taki, jak nam się wydaje. Nasz początkowy model może stać się nieaktualny.

Na przykład firmie Webvan nie udało się zbudować biznesu opartego na sprzedaży artykułów spożywczych przez sieć. Mieli niemal 400 milionów $ kapitału inwestycyjnego i szybko zbudowaną infrastrukturę do wspierania (jak sądzili) dobrze rozwijającego się biznesu. Niestety wszystko okazało się klapą ze względu na koszt dostarczania jedzenia i przeszacowany rynek klientów chcących kupować artykuły spożywcze przez Internet. Ze wszech miar udało im się napisać oprogramowanie i zbudować biznes, ale rynek nie był jeszcze na to gotowy i szybko zbankrutowali. Obecnie znaczna część zbudowanej wtedy infrastruktury jest wykorzystywana przez Amazon.com do wspierania AmazonFresh.

> W teorii praktyka i teoria są tym samym. W praktyce nie są.
>
> – *Albert Einstein*

Obecnie jesteśmy w momencie, gdy teoretycznie możemy poprawnie pisać oprogramowanie i będzie ono działać, ale pisanie odpowiedniego oprogramowania jest znacznie bardziej skomplikowanym problemem. Tutaj przydaje się uczenie maszynowe.

Pisanie odpowiedniego oprogramowania przy zastosowaniu uczenia maszynowego

W książce *The Knowledge-Creating Company* Nonaka i Takeuchi zarysowali, co sprawiło, że japońskie firmy odnosiły takie sukcesy w latach 80-ych XX wieku. Zamiast stosować podejście „od ogółu do szczegółu" przy rozwiązywaniu problemów, uczyły się i zdobywały wiedzę. Pochodzący z tej książki przykład z zagniataniem ciasta na chleb i opracowaniem

urządzenia do wypieku chleba jest doskonałym przykładem iteracji, który można łatwo zastosować w opracowywaniu oprogramowania.

Możemy pójść jeszcze dalej, korzystając z uczenia maszynowego.

Czym dokładnie jest uczenie maszynowe?

Według większości definicji uczenie maszynowe jest zbiorem algorytmów, technik i branżowych sztuczek, które pozwalają maszynom uczyć się na podstawie danych – czyli czegoś, co można przestawić w formie liczbowej (macierze, wektory, itd.).

Aby jednak lepiej zrozumieć uczenie maszynowe, przyjrzyjmy się, jak powstało. W latach 50-ych XX wieku prowadzono szeroko zakrojone badania nad grą w warcaby. Wiele spośród tych modeli skupiało się na lepszej grze i wymyśleniu optymalnych strategii. Moglibyśmy pewnie obecnie stworzyć prosty program grający w warcaby, cofając się od sytuacji wygrywającej, rozrysowując drzewo decyzyjne i optymalizując posunięcia.

Jest to jednak dość zawężony i dedukcyjny sposób rozumowania. Trzeba by w efekcie zaprogramować agenta podejmującego decyzje. W większości tych wczesnych programów nie było zaprogramowanego kontekstu albo irracjonalnych zachowań.

Niemal 30 lat później uczenie maszynowe zaczynało się gwałtownie rozwijać. Wiele tęgich umysłów zaczęło pracować nad problemami obejmującymi filtrowanie spamu, klasyfikowanie i ogólną analizę danych.

Ważną zmianą jest tutaj przeniesienie nacisku z komputerowej dedukcji na komputerową indukcję. Jak w przypadku Sherlocka Holmesa, dedukcja obejmuje zastosowanie skomplikowanych modeli logicznych do uzyskania wniosków. Z kolei indukcja obejmuje przyjęcie danych za prawdziwe i próbę dopasowania modelu do tych danych. Zmiana ta przyczyniła się do wielu ważnych postępów w znajdywaniu dobrych rozwiązań dla typowych problemów.

Problemem przy rozumowaniu indukcyjnym jest jednak to, że możemy algorytmowi dostarczać jedynie dane, które *znamy*. Ilościowe określanie niektórych kwestii jest niezwykle trudne. W jaki sposób na przykład wyrazić liczbowo, jak milutko wygląda kotek na zdjęciu?

W ostatnich 10 latach jesteśmy świadkami renesansu dogłębnego uczenia, które pomaga złagodzić te problemy. Zamiast polegać na danych zakodowanych przez ludzi, algorytmy są w stanie znajdować punkty danych, których wcześniej nie byliśmy w stanie określać liczbowo.

Brzmi to świetnie, ale możliwości te wiążą się z niezwykle wysokimi kosztami i odpowiedzialnością.

Wysoko oprocentowany dług uczenia maszynowego

W artykule opublikowanym niedawno przez Google i zatytułowanym „Machine Learning: The High Interest Credit Card of Technical Debt" (Uczenie maszynowe: dług techniczny

z wysoko oprocentowanej karty kredytowej)[14], Sculley i in. wyjaśniają, że projekty uczenia maszynowego również cierpią na problemy związane z długiem technicznym (tabela 1-1).

Zauważyli, że projekty uczenia maszynowego są z natury skomplikowane, mają niejasno określone granice, polegają w dużym stopniu na zależnościach od danych, cierpią z powodu mieszanego kodu na poziomie systemowym i mogą się znacznie zmieniać ze względu na zmiany w świecie zewnętrznym. Wymieniają problemy, które są szczególnie związane z projektami uczenia maszynowego.

Zamiast omawiać poszczególne problemy, pomyślałem, że ciekawe będzie powiązanie naszego wcześniejszego przeglądu zasad SOLID i TDD oraz refaktoringu i zobaczenie, jak odnosi się to do kodu uczenia maszynowego.

Tabela 1-1 *Wysoko oprocentowany dług uczenia maszynowego*

Problem uczenia maszynowego	Przejawia się jako	Naruszenie zasady SOLID
Splątanie	Zmiana jednego czynnika zmienia wszystko	SRP
Ukryte pętle sprzężenia zwrotnego	Wbudowane, ukryte funkcje w modelu	OCP
Niezadeklarowany dług klientów/widoczności		ISP
Niestabilne zależności między danymi	Ulotne dane	ISP
Nie w pełni wykorzystywane zależności między danymi	Niewykorzystywane wymiary	LSP
Kaskada poprawek		*
Kod spajający	Pisanie kodu, który zajmuje się wszystkim	SRP
Gąszcz potoków	Przesyłanie danych przez złożone przepływy zadań	DIP
Ścieżki eksperymentalne	Ślepe ścieżki, które nigdzie nie prowadzą	DIP
Dług konfiguracji	Wykorzystywanie starych konfiguracji dla nowych danych	*
Stałe pułapy w dynamicznym świecie	Mała elastyczność wobec zmian w korelacjach	*
Zmiany korelacji	Modelowanie korelacji zamiast przyczynowości	Specyficzne dla uczenia maszynowego

Zastosowanie zasad SOLID w uczeniu maszynowym

Zasady SOLID, jak pamiętamy, są jedynie wskazówkami przypominającymi nam o pewnych celach przy pisaniu kodu zorientowanego obiektowo. Wiele algorytmów uczenia maszynowego z zasady nie jest obiektowo zorientowana. Są one funkcjonalne, matematyczne i wykorzystują statystykę, ale nie zawsze tak jest. Zamiast myśleć o sprawach w czysto

14 *http://research.google.com/pubs/pub43146.html*

funkcjonalny sposób, możemy dążyć do wykorzystania obiektów dla każdego wektora i macierzy danych.

SRP

W kodzie uczenia maszynowego jednym z największych wyzwań dla programistów jest zdanie sobie sprawy, że kod i dane zależą wzajemnie od siebie. Bez danych algorytm uczenia maszynowego jest bezwartościowy, a bez algorytmu uczenia maszynowego nie wiemy, co zrobić z danymi. Z definicji więc są ze sobą ściśle powiązane. Ta ścisła zależność jest chyba jednym z największych powodów niepowodzeń projektów uczenia maszynowego.

Zależność ta przejawia się w postaci dwóch problemów w kodzie uczenia maszynowego, którymi są splątanie i kod spajający. *Splątanie* jest czasami nazywane zasadą „zmiana czegokolwiek zmienia wszystko". Najprostszym przykładem są prawdopodobieństwa. Jeśli usuniemy z rozkładu jedno prawdopodobieństwo, to wszystkie pozostałe trzeba dopasować. Jest to naruszenie zasady pojedynczej odpowiedzialności (SRP).

Do możliwych strategii przeciwdziałających tym problemom należą izolowanie modeli, analizowanie zależności wymiarowych[15] i techniki regulujące[16]. Wrócimy do tego problemu podczas omawiania modeli bayesowskich i modeli prawdopodobieństw.

Kod spajający jest kodem, który nawarstwia się z czasem w projekcie programistycznym. Jego celem jest zwykle mało eleganckie spajanie dwóch oddzielnych elementów. Zwykle jest to też ten typ kodu, który stara się rozwiązywać wszystkie problemy, zamiast tylko jednego.

Czy badacze zajmujący się uczeniem maszynowym są skłonni to przyznać, czy nie, to często faktyczne algorytmy uczenia maszynowego są same w sobie dość proste. Otaczający je kod stanowi większość projektu. W zależności od stosowanej biblioteki (GraphLab, MATLAB, scikit-learn, R) mają one swoje własne implementacje wektorów i macierzy, do których najczęściej sprowadza się uczenie maszynowe.

OCP

Zasada otwarte-zamknięte (OCP) jest związana z otwieraniem klas na rozszerzanie, ale nie modyfikacje. Jednym z przejawów tego w kodzie uczenia maszynowego jest problem „zmiana czegokolwiek zmienia wszystko". Może to się pojawiać w dowolnym projekcie programistycznym, ale w projektach uczenia maszynowego często ma to postać ukrytej pętli sprzężenia zwrotnego.

Dobrym przykładem ukrytej pętli sprzężenia zwrotnego jest *przewidywanie przestępstw*. W ostatnich kilku latach wielu badaczy pokazało, że algorytmy uczenia maszynowego można stosować do określania, gdzie zostaną dokonane przestępstwa. Wstępne

15 H. B. McMahan i in., „Ad Click Prediction: A View from the Trenches". W *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013*, Chicago, IL, 11 – 14 sierpnia 2013.

16 A. Lavoie i in., „History Dependent Domain Adaptation". W *Domain Adaptation Workshop at NIPS '11*, 2011.

wyniki pokazały, że te algorytmy wyjątkowo dobrze działają. Niestety mają one też swoją ciemną stronę.

Choć gdy te algorytmy mogą pokazać, gdzie mogą wystąpić przestępstwa, naturalnie doprowadzi to do tego, że policja zacznie patrolować częściej te obszary i wykryje tam więcej przestępstw, co spowoduje samoczynne wzmocnienie algorytmu. Może to prowadzić do potwierdzania uprzedzeń lub wpływać na potwierdzanie wstępnie przyjętej tezy, zaś negatywnym efektem ubocznym może być wzmocnienie dyskryminacji pewnych dzielnic lub grup demograficznych.

Ukryte pętle zwrotne są trudne do wykrycia, ale należy je wyszukiwać i eliminować.

LSP

Obecnie niewiele osób mówi o zasadzie podstawienia Liskov (LSP), ponieważ wielu programistów skłania się bardziej ku kompozycji niż dziedziczeniu. Jednak w świecie uczenia maszynowego zasada LSP jest często naruszana. Wiele razy otrzymujemy zbiory danych, dla których nie mamy jeszcze wszystkich odpowiedzi. Czasami te zbiory danych mają tysiące wymiarów.

Uruchamianie algorytmów na tych danych może w istocie naruszać zasadę LSP. Typowym przejawem tego w kodzie uczenia maszynowego są nie w pełni wykorzystywane zależności między danymi. Często otrzymujemy zbiory danych, które zawierają tysiące wymiarów, co może czasem dawać trafne informacje, a czasem nie. Nasze modele mogą wykorzystywać wszystkie wymiary, ale niektóre z nich w mniejszym stopniu. Na przykład w klasyfikowaniu grzybów jako trujących lub jadalnych informacja taka jak zapach może być znaczącym wskaźnikiem, a liczba pierścieni nie. Liczba pierścieni ma małą ziarnistość i może wynosić tylko zero, jeden lub dwa; dlatego w istocie nie wpływa znacząco na nasz model klasyfikowania grzybów. Możemy więc wyciąć tę informację z naszego modelu i nie spowoduje to znacznego spadku jego skuteczności.

Można by się zastanawiać, dlaczego jest to związane z zasadą LSP, a powodem jest to, że jeśli możemy korzystać jedynie z najmniejszego zbioru punktów danych (lub funkcji), budujemy najlepszy możliwy model. Jest to też zgodne z brzytwą Ockhama, która stwierdza, że najprostsze rozwiązanie jest najlepsze.

ISP

Zasada rozdzielenia interfejsów (ISP) opiera się na pojęciu, że interfejs specyficzny dla klienta jest lepszy niż interfejs ogólnego zastosowania. W projektach uczenia maszynowego może być to często trudne do wymuszenia ze względu na ścisłe powiązanie danych z kodem. W kodzie uczenia maszynowego zasada ISP jest zwykle naruszana przez dwa typy problemów: *dług widoczności* i *niestabilne dane*.

Weźmy na przykład pod uwagę sytuację, gdy firma ma raportową bazę danych, wykorzystywaną do zbierania informacji o sprzedaży, dostawach i innych ważnych danych. Wszystko jest zarządzane przez jakiegoś rodzaju projekt, który wprowadza te dane do bazy danych. Klient, wykorzystujący tę bazę danych, definiuje projekt uczenia maszynowego,

który wykorzystuje poprzednie dane dotyczące sprzedaży do przewidywania sprzedaży w przyszłości. Pewnego dnia ktoś zmienia nazwę tabeli, której nazwa była myląca, na coś bardziej przydatnego. Nagle wszystko przestaje działać i wszyscy zastanawiają się, co się stało.

Okazuje się, że projekt uczenia maszynowego nie był jedynym klientem, wykorzystującym te dane; sześć innych baz danych było też z nimi powiązanych. Fakt, że istniało tylu niezadeklarowanych klientów danych, jest sam w sobie elementem długu dla projektu uczenia maszynowego.

Ten typ długu jest zwany długiem widoczności, a choć najczęściej nie wpływa na stabilność projektu, to czasami może wstrzymywać jego dalszy rozwój.

Dane są zależne od kodu używanego do przeprowadzania indukcji na ich podstawie, więc budowanie stabilnych projektów wymaga stabilnych danych. Bardzo często tak nie jest. Weźmy na przykład cenę akcji; rano może być wysoka, ale po kilku godzinach może znacząco spaść.

Narusza to zasadę ISP, ponieważ patrzymy na ogólny strumień danych zamiast na dane szczególne dla klienta, co może utrudniać zbudowanie odpowiednich algorytmów. Często budowany jest jakiś schemat wag dla danych albo definiowane są wersje dla strumieni danych. Ten schemat wersjonowania jest sposobem na ograniczenie ulotności przewidywań modelu.

DIP

Zasada odwrócenia zależności (DIP) jest związana z ograniczaniem nagromadzenia danych i zwiększaniem elastyczności kodu pod kątem przyszłych zmian. W projekcie uczenia maszynowego elementy konkretne mogą pojawiać się na dwa szczególne sposoby: *gąszcz potoków* i *ścieżki eksperymentalne*.

Gąszcz potoków występuje w projektach sterowanych danymi i jest zwykle formą kodu spajającego. Oznacza to przygotowywanie i przenoszenie mieszaniny danych. W niektórych przypadkach kod ten wiąże wszystko razem, aby model mógł działać na spreparowanych danych. Niestety ten gąszcz z czasem staje się bardziej skomplikowany i trudniejszy w użyciu.

Kod uczenia maszynowego wymaga zarówno oprogramowania, jak i danych. Są one ze sobą powiązane i nierozdzielne. Czasami musimy testować działanie podczas produkcji. Niekiedy testy na komputerach nieprodukcyjnych dają nam fałszywą nadzieję. Ścieżki eksperymentalne nawarstwiają się z czasem i w końcu zanieczyszczają nasz obszar roboczy. Najlepszym sposobem na ograniczanie związanego z tym długu jest wprowadzenie starej techniki z języka C, która polega na oznaczaniu czegoś jako gotowego do usunięcia. Jeśli dana metoda jest wywoływana w środowisku produkcyjnym, będzie rejestrować zdarzenie w pliku dziennika, który można później wykorzystać do oczyszczenia bazy kodu.

Jeśli ktoś zajmował się wcześniej odśmiecaniem danych, to pewnie słyszał o tej metodzie zwanej oznacz i zamieć. Najpierw oznaczamy obiekt jako gotowy do usunięcia, a następnie wymiatamy (usuwamy) zaznaczone obiekty.

Kod uczenia maszynowego jest skomplikowany

Czasami kod uczenia maszynowego może być trudny do napisania i zrozumienia, ale na pewno nie jest to niemożliwe. Pamiętając o analogii lotniczej, od której zaczęliśmy, możemy wykorzystywać wskazówki SOLID jako startową listę kontrolną przy pisaniu kodu uczenia maszynowego.

Analogicznie możemy porównać kod uczenia maszynowego do lotu statkiem kosmicznym – nie jest to coś niebywałego, ale jest znacznie trudniejsze od lotu samolotem. Stosując listę kontrolną SOLID, możemy skutecznie uruchamiać nasz kod z wykorzystaniem TDD i refaktoringu. Podsumowując, pisanie udanego kodu uczenia maszynowego sprowadza się do utrzymywania dyscypliny zachowującej zasady projektowe, które przedstawiliśmy w tym rozdziale oraz pisania testów wspierających hipotezy oparte na naszym kodzie. Innym krytycznym elementem pisania skutecznego kodu jest elastyczność i dostosowywanie się do zmian, które napotkamy w rzeczywistym świecie.

TDD: metoda naukowa 2.0

Każdy prawdziwy naukowiec jest marzycielem i sceptykiem. Wysłanie człowieka na księżyc było śmiałym pomysłem, ale zrealizowaliśmy go dzięki systematycznym badaniom i rozwojowi. Tak samo jest z kodem uczenia maszynowego. Niektóre zastosowania są fascynujące, ale przy tym trudne do zrealizowania.

Tajemnica tkwi w zastosowaniu listy kontrolnej SOLID w uczeniu maszynowym oraz narzędzi TDD i refaktoringu.

TDD jest bardziej stylem rozwiązywania problemów, a nie odgórnym nakazem. Testowanie daje nam pętlę sprzężenia zwrotnego, którą możemy wykorzystać do pracy nad trudnymi problemami. Naukowcy potwierdzą, że najpierw muszą sformułować hipotezę, następnie opracować ją teoretycznie i przetestować. Jako praktycy TDD możemy stwierdzić, że podobnie działa proces: czerwone (test nieudany), zielone (test udany), refaktoring.

Ta książka będzie głęboko wnikać w zastosowanie TDD i zasad SOLID w uczeniu maszynowym, prowadząc do refaktoringu kodu w celu uzyskania stabilnego, skalowalnego i łatwego w użyciu modelu.

Refaktoring wiedzy

Jak wspominaliśmy, refaktoring jest możliwością edycji naszej dotychczasowej pracy i przemyślenia wcześniejszych ustaleń. W książce tej omówimy refaktoring typowych problemów z uczeniem maszynowym w odniesieniu do poszczególnych algorytmów.

Plan tej książki

W książce tej zamierzam omówić wiele zagadnień dotyczących uczenia maszynowego. Na koniec powinniśmy lepiej rozumieć, jak pisać kod uczenia maszynowego, a także jak wdrażać go w środowisku produkcyjnym i skalować. Uczenie maszynowe jest fascynującą dziedziną, która pozwala nam wiele osiągnąć, ale bez odpowiedniej dyscypliny, list kontrolnych i wskazówek wiele projektów uczenia maszynowego będzie skazanych na porażkę.

W tej książce będziemy wracać do omówionych w tym rozdziale zasad SOLID, testowania kodu (przy użyciu różnych środków) oraz refaktoringu jako sposobów na stałe poprawianie wydajności naszego kodu.

Każdy rozdział będzie objaśniał wykorzystywane pakiety języka Python i opisywał ogólny plan testów. Dla kodu uczenia maszynowego możemy pisać testy pomagające nam w zrozumieniu danego problemu.

Rozdział 2

Szybkie wprowadzenie do uczenia maszynowego

Skoro ktoś sięgnął po tę książkę, to pewnie interesuje się uczeniem maszynowym. Zagadnienie to jest często definiowane dosyć ogólnikowo. W tym szybkim wprowadzeniu omówię, czym dokładnie jest uczenie maszynowe i przedstawię ogólne ramy algorytmów uczenia maszynowego.

Czym jest uczenie maszynowe?

Uczenie maszynowe jest połączeniem teoretycznie solidnej informatyki z praktycznie chaotycznymi danymi. Chodzi zasadniczo o interpretowanie danych przez maszyny w sposób zbliżony do tego, jak robią to ludzie.

Uczenie maszynowe jest rodzajem sztucznej inteligencji, zgodnie z którym algorytm lub metoda wyciąga jakieś wzorce ze zbioru danych. Uczenie maszynowe rozwiązuje kilka ogólnych problemów – są one wymienione w tabeli 2-1 i opisane w dalszej części rozdziału.

Tabela 2-1 *Problemy, które mogą być rozwiązane przez uczenie maszynowe.*

Problem	Kategoria uczenia maszynowego
Dopasowywanie jakichś danych do funkcji lub aproksymacja funkcji	Uczenie nadzorowane
Ustalanie, czym są dane, bez żadnego sprzężenia zwrotnego	Uczenie nienadzorowane
Maksymalizacja korzyści w czasie	Uczenie wzmacniane