Jurnal Ilmu Teknik dan Informatika (TEKNIK), Vol.3, No.1, Mei 2023, pp. 55 - 72
p-ISSN : 2808-8751 (print)
e-ISSN : 2798-2513 (online)
DOI: https://doi.org/10.51903/teknik.v3i1
http://journal.stiestekom.ac.id/index.php/TEKNIK

Analisis Tes Flakiness Untuk Machine Learning Menggunakan Python

Silvia FardilaSoliha[1]

[1]Universitas STEKOM
Jl. Majapahit No.605, Pedurungan, Semarang, Jawa Tengah Telp. (024) 6723456
e-mail : Silvia@yahoo.com

ARTICLE INFO

Article history:

Received 10 maret 2023
Received in revised form 25 maret 2023
Accepted 1 april 2023
Available online Mei 2023

ABSTRAK

Tujuan Utama: Tujuan dari penelitian ini adalah untuk mengetahui penyebab dari flaky test yang paling umum terjadi pada proyek Python dengan membandingkan penelitian yang berfokus pada proyek Java sebelumnya. Background problem: Tes tidak stabil dapat gagal atau lulus tanpa ada perubahan pada kode yang diuji, ini dapat menghancurkan kepercayaan pengembang pada rangkaian pengujian dan jika diabaikan menyebabkan bug dalam kode yang dirilis. Pembaharuan: penelitian menggunakan pendekatan empiris dari proyek Python open-source paling populer di GitHub. Sejumlah 197 komitmen dengan kata kunci yang menunjukkan kelemahan uji diperiksa secara manual dan dikategorikan menurut akar penyebab kelemahan masing-masing. Metode Penelitian: metode analisis pengujian flakiness digunakan dengan urutan tahap filtering commit dan dua analisis. Temuan: hasil penelitian dibandingkan dengan studi proyek Java sebelumnya, dan ditemukan dua penyebab kelemahan yakni presisi dan pelatihan (jaringan Machine learning). Flakiness presisi disebabkan oleh pernyataan dengan ambang batas yang terlalu tinggi atau terlalu rendah. Kelemahan pelatihan disebabkan oleh pengaturan pelatihan yang salah dari jaringan Machine learning dalam pengujian. Sebagian besar tes dalam proyek Python ditemukan tidak stabil karena masalah dengan menunggu asinkron, presisi, dan jaringan. Kesimpulan: Developer Python dimasa depan akan mendapat manfaat dari pengetahuan tentang jebakan umum yang dapat menyebabkan kelemahan dalam rangkaian pengujian mereka. Hasil dari penelitian ini dapat digunakan sebagai referensi bagi para peneliti di masa depan dengan area penlitian tes flakiness atau area serupa lainnya..

Kata Kunci: *Flakiness Test, Machine Learning, Phyton*

1. PENDAHULUAN

Bahasa Python

Bahasa pemrograman Python adalah salah satu bahasa paling populer di dunia (SODS, (2019)). Ini adalah bahasa yang ditafsirkan yang diketik secara dinamis. Global Interpreter Lock (GIL) Python, (2017) adalah

Received 10 maret 2023; Revised 25 maret 2023; Accepted 1 april 2023

p-ISSN: 2808-8751 e-ISSN : *2798-2513*

mutex yang mencegah bytecode Python dieksekusi oleh banyak utas. Ini mencegah program Python multithread mengambil keuntungan penuh dari sistem multiprosesor dalam situasi tertentu. Operasi yang berjalan lama atau pemblokiran seperti IO atau pemrosesan gambar terjadi di luar GIL. Semua utas lainnya perlu mendapatkan GIL sebelum dapat dieksekusi. Ini diperlukan karena manajemen memori Python tidak aman untuk thread.

Pengujian Regresi

Pengujian regresi adalah cara melakukan pengujian ulang untuk memastikan bahwa kode yang sudah diuji masih berfungsi setelah perubahan (Anirban, (2015). Perubahan dapat berupa peningkatan perangkat lunak seperti fitur baru, atau perbaikan bug. Jika rangkaian uji regresi tidak terlalu besar, semua pengujian dapat dijalankan untuk setiap perubahan baru. Jika test suite memakan waktu terlalu lama untuk dijalankan, subset dari test suite harus dipilih dengan RTS (regression test selection). Ada beberapa teknik RTS yang tersedia (Engström et al., (2009).

Integrasi Berkelanjutan

Sistem CI (Continuous Integration) adalah sistem yang melakukan kompilasi, pembangunan, dan pengujian perangkat lunak. Sistem CI terhubung ke sistem kontrol versi, dapat mendeteksi dan memulai pembangunan CI baru ketika permintaan komit, cabang, atau tarik baru ditambahkan. Hasil build CI sering dilaporkan melalui email dan ditampilkan dalam histori komit atau diskusi pull request dengan integrasi dengan platform hosting kode seperti GitHub. Sistem CD (Continuous Deployment) akan menyebarkan kode ke lingkungan produksi secara otomatis. Dalam sebuah survei besar dari tahun 2018 oleh perusahaan komputasi awan Digital Ocean (Digital Ocean, (2018) sekitar 58% responden mengatakan bahwa mereka menggunakan proses integrasi berkelanjutan. Untuk organisasi yang lebih besar (>100 karyawan) menggunakan CI atau CD menjadi praktik standar sekitar dua pertiga menyatakan mereka memiliki proses integrasi berkelanjutan dan lebih dari 50% juga menggunakan penerapan berkelanjutan.

Falky tes mempengaruhi integrasi berkelanjutan

Memasukkan Flaky tes ke dalam karantina tidak boleh terlalu lama karena akan merusak sistem deteksi bug. Microsoft menerapkan sistem di mana pengujian yang tidak stabil terdeteksi dengan menjalankan pengujian beberapa kali (Microsoft, (2017). Jika tes ditemukan tidak stabil, maka tes tersebut akan dikeluarkan dari rangkaian tes dan diajukan sebagai bug reliabilitas secara oromatis. Hilton et al., (Hilton et al., (2017) menyimpulkan bahwa mengidentifikasi pengujian yang tidak pasti dapat membantu pengembang yang menggunakan sistem CI memahami kegagalan CI akibat pengujian yang tidak pasti.

Flaky Tes

Penelitian sebelumnya menunjukkan bahwa sebagian besar pengembang menganggap tes dengan hasil deterministik adalah ide yang bagus, karena non-determinisme membuat sulit untuk mengandalkan output dari hasil tes. Dalam studi sebelumnya sejumlah penyebab umum dari Flaky Tes telah ditemukan (Luo et al., (2014); Palomba & Zaidman, (2018); Vahabzadeh e al., (2015); Thorve et al., (2018)) dan beberapa penyebab paling umumnya adalah Menunggu Asinkron dan Konkurensi. Karena menunggu asinkron flakiness akan terjadi ketika kode yang dieksekusi memulai panggilan asinkron dan tidak menunggu hasil dari panggilan yang ada. Flakiness terkait konkurensi terjadi karena utas atau proses yang berbeda berinteraksi dengan cara yang buruk (Luo et al., (2014); Palomba & Zaidman, (2018); Vahabzadeh e al., (2015); Thorve et al., (2018). Kelemahan ketergantungan urutan pengujian terjadi jika pengujian perangkat lunak bersifat independen satu sama lain dan tidak diisolasi. Untuk memastikan bahwa status satu pengujian tidak memengaruhi pengujian lainnya, setiap pengujian sebaiknya mengatur dan membersihkan statusnya sendiri. Asumsi independensi tes dipelajari oleh Zhang et al., (Zhang et al., (2014) yang menemukan bahwa sulit bagi programmer untuk mengidentifikasi ketergantungan tes. Teknik seperti penentuan prioritas tes (Luo et al., (2018); Khatibsyarbini et al., (2018)) dan pemilihan tes (Saber et al., (2018); Garousi et al., (2018) bergantung pada independensi tes dalam rangkaian tes. Sebagian besar tes flakiness yang dikategorikan dengan IO menjadi penyebab kelemahan, tes ini memiliki masalah dalam pembacaan file.

Beberapa teknik telah dikembangkan untuk mendeteksi tes flakiness. Bel et al., (Bell et al., (2018) mengembangkan DeFlaker tools yang mampu mendeteksi pengujian yang tidak stabil dalam proyek Java. Cara khas developer untuk mendeteksi pengujian yang tidak stabil adalah dengan menjalankannya kembali beberapa kali dan memperhatikan hasilnya. DeFlaker tool mengecek versiuntuk menentukan kode baru dieksekusi. Dengan menggunakan DeFlaker tool, mereka dapat menemukan 95,5% tes flakiness yang

p-ISSN: 2808-8751 e-ISSN : *2798-2513*

terkonfirmasi dalam sebuah studi kasus. King et al., (King et al., (2018) mengusulkan pendekatan Machine learning yang menggunakan model jaringan Bayesian untuk memprediksi kegagalan karena lingkungan, atau pengujian yang tidak stabil. Dalam studi terbaru oleh Lam et al., (Lam et al., (2019) kerangka kerja (iDFlakies) dikembangkan untuk mendeteksi dan mengklasifikasikan beberapa jenis tes flakiness. Framework ini terbukti mampu mendeteksi tes yang tidak stabil dan mengklasifikasikannya sebagai OD (order-dependent) atau NOD (non-order-dependent). Oleh karena itu, kerangka kerja dapat mendeteksi jika suatu pengujian memiliki ketergantungan urutan pengujian. Framework ini diuji pada sejumlah besar proyek populer dari GitHub. Popularitas ditentukan menggunakan bintang proyek GitHub. Dengan kerangka ini Lam et al., (Lam et al., (2019) membangun satu set data dari 422 tes flakiness. 50,5% dari pengujian dalam kumpulan data tidak stabil karena urutan pengujian dijalankan dan 49,5% sisanya karena alasan lain.

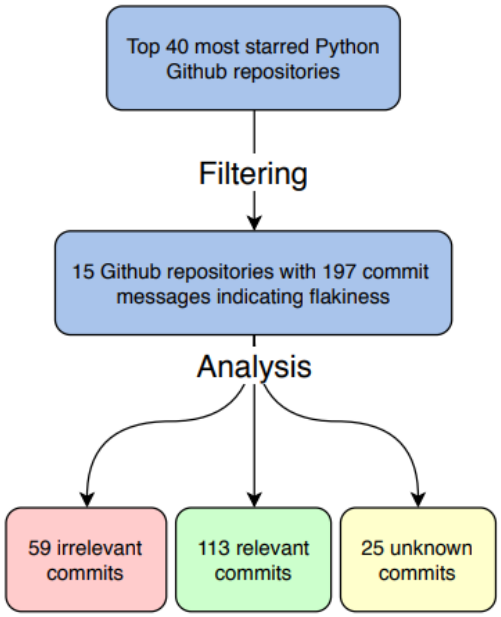
Luo et al., (2014) dan Thorve et al., (2018) memiliki metodologi yang sangat mirip dengan penelitian ini secara manual dalam menentukan kumpulan data komit yang ditemukan dari pencarian kata kunci dalam pesan komit. Lam et al., (2019) menggunakan bintang GitHub untuk menemukan proyek open source populer seperti penelitian ini. Studi ekstensif pertama tentang tes flakiness yang dipublikasikan oleh Luo et al., (2014) menganalisis repositori proyek ASF (Apache Software Foundation). Para peneliti mengumpulkan data dengan mencari pesan-pesan komit yang mengandung kata kunci "flak" (untuk memasukkan flaky, flakiness, dll.) dan "in flaks" (untuk memasukkan intermiten, intermiten, dll.). sejumlah 201 komitmen dari 51 proyek ASF diperiksa secara manual untuk memperbaiki pengujian yang tidak stabil dengan tujuan untuk menemukan akar penyebab kelemahan pengujian. Studi ini menyatakan akar penyebab paling umum dari flakiness berasal dari menunggu asinkron, konkurensi, dan ketergantungan urutan pengujian. Studi ini juga mengidentifikasi pendekatan yang dapat memanifestasikan perilaku tidak stabil dan menjelaskan strategi umum yang digunakan developer untuk memperbaiki pengujian tidak stabil. Vahabzadeh e al., (2015) melakukan studi empiris bug dalam kode uji pada tahun 2015. Repositori bug dan sistem kontrol versi dari semua proyek ASF yang tersedia pada saat itu (total 211) ditambang untuk data. Ditemukan sejumlah 5556 laporan bug terkait tes dan 443 di antaranya diambil sampelnya secara acak untuk studi kualitatif. Studi kualitatif menemukan bahwa setengah dari proyek memeriksa bug termanifestasi dalam kode pengujian mereka, tetapi sebagian besar bug adalah alarm palsu. Sejumlah 21% dari bug tes alarm palsu disebabkan oleh tes yang tidak stabil.

Palomba & Zaidman, (2018) mengklasifikasikan tes flakiness dengan kategori yang ditentukan oleh Luo et al., (2014). Mereka menemukan bahwa asynchronous waiting, IO dan concurrency adalah penyebab paling umum dari kelemahan dalam pengujian. Para peneliti percaya bahwa mereka menemukan penyebab yang berbeda Luo et al., (2014) karena mereka menggunakan kumpulan data yang lebih besar. Ahmad et al., (2019) meniliti berbagai faktor yang mempengaruhi tes flakiness dan menemukan 23 faktor unik yang meningkatkan, menurunkan atau mempengaruhi kemampuan untuk menemukan kelemahan uji. Beberapa faktor yang meningkatkan tes flakiness adalah sistem yang rumit, bau uji, kurangnya pemahaman lingkungan, kasus uji lama, dan kurangnya persyaratan yang jelas. Thorve et al., (2018) mempelajari tes flakiness dalam proyek Android. Mereka mencari GitHub untuk proyek Android open source untuk menemukan komitmen yang berhubungan dengan kelemahan pengujian. Metodologi yang terlibat mencari pesan komit dengan kata kunci yang mirip dengan Luo et al., (2014), 77 komit diperiksa secara manual dan diurutkan berdasarkan penyebab kelemahannya. Ditemukan tiga penyebab baru yang sebelumnya tidak dilaporkan oleh penelitian yang diterbitkan, Ketergantungan (pada perangkat keras, perpustakaan, OS), logika program (pengujian tidak memahami perilaku program dengan benar) dan UI. Tiga penyebab paling umum dari kelemahan adalah Concurrency (termasuk menunggu asinkron), ketergantungan, dan logika program

2. METHODOLOGY

Berbeda dari studi sebelumnya yang sebagian besar proyek menggunakan Java, studi ini menggunakan sejumlah proyek Python open source populer dan mempelajari riwayat versinya. Tujuannya adalah untuk menemukan komit yang mencoba memperbaiki pengujian yang tidak stabil. Komit tersebut kemudian diperiksa untuk melihat apa yang menyebabkan flakiness tersebut. Metodologi analisis pengujian flakiness telah berhasil digunakan oleh penelitian lain tentang pengelupasan dalam pengujian.

p-ISSN: 2808-8751 e-ISSN : *2798-2513*



3.

4. **Gambar 1. Kerangka Kerja Penelitian**

Penyaringan Komit

Tahap pertama penelitian ini difokuskan pada pengumpulan data yang relevan untuk digunakan dalam penelitian dengan data data yang dikumpulkan bersifat beragam termasuk berbagai jenis proyek. Pada fase pertama, komit yang menunjukkan beberapa kelemahan disaring dari proyek Python populer. Itu menghasilkan kumpulan data dari 197 komitmen. Pada fase kedua, komit yang difilter dianalisis dan dikategorikan dengan kategori kelemahan. Setelah tahap analisis, 59 dari 197 komit dibuang karena tidak relevan, karena tidak terkait langsung dengan tes flakiness. Sejumlah 25 komitmen dibuang, karena terlalu sulit untuk dipahami dan diklasifikasi. 113 komit yang tersisa semuanya dianggap relevan, karena berisi perbaikan, atau menonaktifkan tes tidak stabil yang akar masalahnya mungkin untuk dikategorikan. Sejumlah 113 komitmen yang relevan ini adalah yang digunakan dalam hasil, dan kesimpulan diambil darinya. Sejumlah 12 dari komit yang relevan hanya menonaktifkan tes flakiness yang bersangkutan.

Analisis Flakiness di Commits

Tujuan dari analisis ini adalah untuk memeriksa semua dari 197 komitmen yang dipilih dari fase penyaringan untuk menentukan relevansi komit. Komit yang berisi perbaikan untuk pengujian yang tidak pasti atau menonaktifkan pengujian yang tidak pasti dianggap relevan. Semua komitmen lainnya diabaikan karena tidak relevan. Selain itu analisi ini juga untuk menentukan apa yang menyebabkan flakiness tes dan memilih kategori penyebab flakiness untuk semua komitmen yang relevan. Langkah pertama dalam proses analisis adalah meninjau pesan komit dan perubahan kode. Jika sulit untuk menentukan penyebab kelemahan hanya berdasarkan pesan komit dan perubahan kode, setiap masalah yang terkait atau permintaan penarikan diperiksa. Terkadang komit berisi nomor laporan bug untuk sistem laporan bug terpisah (misalnya CPython dan Django). Jika komit yang sedang dianalisis tidak mengandung perbaikan untuk pengujian yang tidak stabil maka laporan bug dan komit sebelumnya atau yang akan datang akan diperiksa untuk melihat dan menemukan masalah yang muncul pada komit asli dapat ditemukan. Jika tidak ada yang dapat ditemukan akan dilabeli tidak relevan. Jika ditemukan komit asli yang memperkenalkan perbaikan maka komit dengan permintaan gabungan akan diabaikan. Hampir 40% dari komitmen yang tidak relevan adalah duplikat dan sekitar 30% tidak terkait dengan pengujian sama sekali. Komitmen tidak relevan lainnya adalah gabungan dari logging debug atau pengujian tidak stabil yang tidak terkait dengan kode Python, dan terkadang perubahan yang tidak signifikan untuk memicu proses pipeline CI baru. Beberapa proyek memiliki komitmen yang cukup besar yang pada akhirnya harus dibuang. Untuk CPython misalnya, hampir setengah dari komitmen tidak relevan dengan penelitian.

Pemilihan Kategori Penyebab Flakiness

Selama fase analisis, semua komitmen yang relevan diberi kategori penyebab kelemahan. Kategori yang digunakan mula-mula berdasarkan kategori dari penelitian serupa sebelumnya (Luo et al., (2014); Palomba

p-ISSN: 2808-8751 e-ISSN : *2798-2513*

& Zaidman, (2018); Vahabzadeh e al., (2015); Thorve et al., (2018)). Banyak komit yang cocok dengan kategori yang digunakan oleh penelitian serupa sebelumnya. Tetapi untuk beberapa komitmen, tidak ada kategori yang masuk akal dan beberapa kategori baru dibuat.

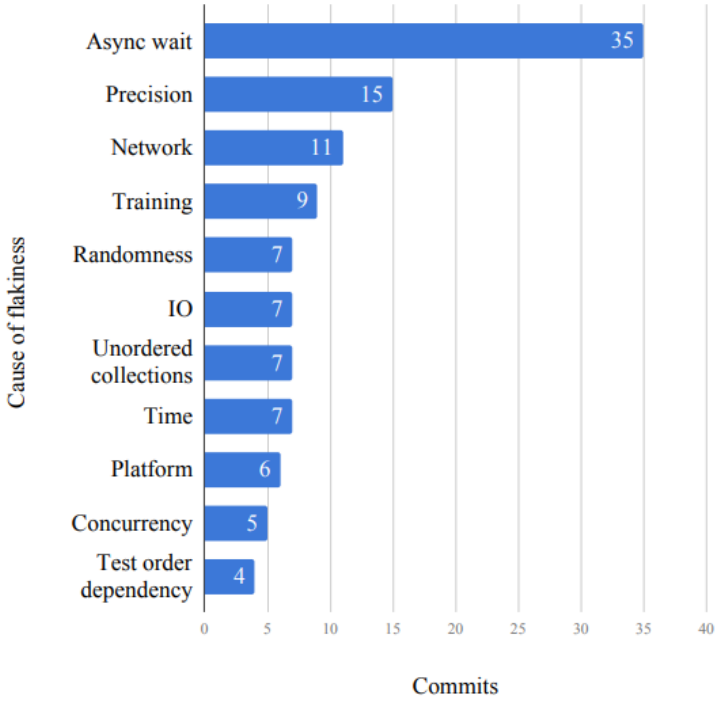
IMPLIKASI

Penyebab Flaky Tes

Tabel 2 dan gambar 3 menunjukkan jumlah dan distribusi dari penyebab uji flaky yang ditemukan. Tiga penyebab paling umum adalah masalah dengan asyn chronous waiting, precision, dan network. Beberapa contoh kode menyoroti perubahan yang dilakukan pada warna. Warna merah berarti kode telah dihapus dan warna hijau menunjukkan bahwa kode telah ditambahkan.

Tabel 2 Distribusi penyebab flakiness

Flakiness cause	Share	#Commits
Async wait	31%	35
Precision	13%	15
Network	10%	11
Training	8%	9
Randomness	6%	7
IO	6%	7
Unordered collections	6%	7
Time	6%	7
Platform	5%	6
Concurrency	4%	5
Test order dependency	4%	4



Gambar 3 Distribusi Penyebab Flakiness

Asynchronous Waiting (async wait) 35 dari 113 (31%) komit berasal dari kategori async wait dan memiliki tes yang tidak stabil karena asynchronous waiting. Komit diberi label dengan kategori ini saat pengujian membuat panggilan asinkron dan tidak menunggu dengan benar hingga hasil panggilan tersedia sebelum mulai menggunakannya. Sebuah contoh dapat dilihat pada gambar 4 di mana tes selenium1 terkelupas dari

p-ISSN: 2808-8751 e-ISSN : *2798-2513*

proyek Django ditampilkan. Kelemahannya berasal dari tidak menunggu dengan benar untuk bidang kata sandi dibuat sebelum mencoba mengaksesnya. Untuk memitigasi masalah ini, penantian pembuatan bidang kata sandi ditambahkan pada baris 11. Cara umum untuk memitigasi masalah penantian asinkron adalah dengan menggunakan panggilan wait_for seperti pada contoh Django yang dijelaskan di atas. Panggilan seperti itu memblokir eksekusi hingga sumber daya yang ditentukan tersedia, kondisi lain terpenuhi, atau batas waktu atas tercapai. Jika pernyataan tidur digunakan alih-alih wait_for, utas akan membuang waktu menunggu, meskipun sumber daya sudah tersedia. Jika tidak ada kemungkinan bawaan dengan pustaka atau kerangka kerja tertentu yang digunakan untuk menggunakan wait_for, ada banyak cara untuk membuatnya sendiri. Paket pythons threading2 dan asyncio3 memiliki objek Event yang dapat digunakan untuk menunggu kondisi yang ditentukan. Mengumpulkan status sumber daya dengan tidur singkat juga merupakan solusi yang lebih baik daripada menggunakan tidur panjang.

```
1  def test_ForeignKey_using_to_field(self):
2      self.admin_login(username='super', password='secret', login_url='/')
3      self.selenium.get('%s%s' % (
4          self.live_server_url,
5          reverse('admin:admin_widgets_profile_add')))
6      main_window = self.selenium.current_window_handle
7      # Click the Add User button to add new
8      self.selenium.find_element_by_id('add_id_user').click()
9      self.wait_for_popup()
10     self.selenium.switch_to.window('id_user')
11     self.wait_for("#id_password")
12     password_field = self.selenium.find_element_by_id('id_password')
13     password_field.send_keys('password')
```

Gambar 4 Bagian dari uji selenium tidak stabil dengan masalah menunggu asinkron, dari proyek Django

Presisi

Dejumlah 15 dari 113 (13%) komit diberi label dengan kategori presisi. Kelemahan berasal dari pernyataan yang terlalu sulit dicapai, atau menggunakan rentang nilai valid yang terlalu sempit. Misalnya, ini dapat berupa pernyataan yang memeriksa keakuratan jaringan Machine learning yang telah dilatih oleh kode yang diuji. Flaky Tes dari proyek Keras dapat dilihat pada gambar 5. Tes ini memeriksa apakah pengoptimal harus mencapai akurasi target > 0,9. Tes memiliki perilaku tidak stabil yang diselesaikan dengan menurunkan target untuk memungkinkan akurasi ≥ 0,89. Untuk mengurangi masalah presisi, kehati-hatian harus diberikan saat merancang pernyataan pengujian. Ini sangat penting ketika kode yang diuji bersifat non-deterministik. Pengembang sebaiknya mengevaluasi interval atau ambang beberapa kali secara lokal sebelum melakukan pengujian yang memiliki potensi kesalahan presisi.

```
1  def _test_optimizer(optimizer, target=0.9  0.89):
2      model = get_model(X_train.shape[1], 10, y_train.shape[1])
3      model.compile(loss='categorical_crossentropy',
4                    optimizer=optimizer,
5                    metrics=['accuracy'])
6      history = model.fit(X_train, y_train, nb_epoch=12,
7                  batch_size=16,
8                  validation_data=(X_test, y_test),
9                  verbose=2)
10     config = optimizer.get_config()
11     assert type(config) == dict
12     assert history.history['val_acc'][-1] > >= target
```

Gambar 6 Flaky test dengan masalah presisi, dari proyek Keras

Jaringan

Sejumlah 11 dari 113 (10%) komit memiliki kelemahan yang disebabkan oleh ketergantungan pada interaksi dengan jaringan. Contoh umum dari hal ini adalah tidak menangani kesalahan dengan benar dalam respons jaringan. Gambar 6 menunjukkan bagian dari pengujian CPython yang tidak stabil karena koneksi jaringan terkadang gagal. Dalam hal ini pengembang melawan kelemahan dengan menambahkan for loop untuk mencoba kembali koneksi beberapa kali jika gagal. Tes lain dari CPython yang menguji pustaka telnet mereka juga tidak stabil. Kelemahan dalam kasus ini diselesaikan dengan mengganti semua panggilan jaringan dengan mock dan stub (Thomas & Hunt, (2002). Perubahan pada commit4 terlalu

p-ISSN: 2808-8751 e-ISSN : *2798-2513*

panjang untuk ditampilkan di sini, tetapi sebagian kecilnya dapat dilihat pada gambar 7. Cara terbaik untuk mengurangi kelemahan jaringan adalah dengan menghilangkan ketergantungan pada jaringan dengan menggunakan mock atau stub Martin, (2011). Python menyertakan perpustakaan tiruan bawaan yang disebut unittest.mock5. Mock digunakan untuk meniru perilaku dependensi atau unit kompleks lainnya. Dengan membuat tiruan yang mengembalikan jawaban yang diharapkan dari panggilan jaringan, flakiness yang disebabkan oleh masalah jaringan dapat dihindari. Tes tidak akan bergantung pada panggilan jaringan lagi. Alih-alih melakukan panggilan kerja jaringan atau operasi lain, respons yang diharapkan harus dibuat dan dikembalikan secara lokal. Ini akan memberikan tes perilaku yang lebih deterministik untuk menghilangkan ketergantungan eksternal.

Beberapa orang berpikir bahwa tiruan atau rintisan tidak boleh digunakan dalam pengujian fungsional karena mereka percaya bahwa koneksi yang sebenarnya harus diuji Martin, (2011). Tetapi karena pengujian otomatis hanya memiliki sedikit atau tidak ada gunanya jika tidak stabil, tiruan dan rintisan mungkin diperlukan. Untuk memastikan tiruan atau rintisan memiliki representasi yang benar dari layanan eksternal, uji Kontrak (Fowler, (2011) dapat digunakan. Ini adalah pengujian yang harus dijalankan secara berkala dan periksa apakah tiruan atau rintisan memberikan jawaban yang sama dengan layanan eksternal yang seharusnya ditiru. Mock dan stub juga memiliki keuntungan bahwa mereka akan membuat pengujian berjalan lebih cepat karena tidak ada panggilan jaringan atau perhitungan lambat yang harus ditunggu oleh pengujian.

```
...
with test_support.transient_internet():
    f = urllib.urlopen('https://sf.net')
buf = f.read()
f.close()
...
```

Gambar 6 Bagian dari pengujian tidak stabil dengan masalah jaringan, dari proyek CPython

p-ISSN: 2808-8751 e-ISSN : *2798-2513*

```
...
class SocketStub(object):
    ''' a socket proxy that re-defines sendall() '''
    def __init__(self, reads=[]):
        self.reads = reads
        self.writes = []
        self.block = False
    def sendall(self, data):
        self.writes.append(data)
    def recv(self, size):
        out = b''
        while self.reads and len(out) < size:
            out += self.reads.pop(0)
        if len(out) > size:
            self.reads.insert(0, out[size:])
            out = out[:size]
        return out
...

def test_socket(reads):
    def new_conn(*ignored):
        return SocketStub(reads)
    try:
        old_conn = socket.create_connection
        socket.create_connection = new_conn
        yield None
    finally:
        socket.create_connection = old_conn
    return

def test_telnet(reads=[], cls=TelnetAlike):
    ''' return a telnetlib.Telnet object that uses a SocketStub with
        reads queued up to be read '''
    for x in reads:
        assert type(x) is bytes, x
    with test_socket(reads):
        telnet = cls('dummy', 0)
        telnet._messages = '' # debuglevel output
    return telnet
...
```

Gambar 7 Metode helper dalam bentuk stub untuk mengurangi flakiness in Suite pengujian telnet CPython.

Pelatihan

Sejumlah 9 dari 113 (8%) komit ditemukan tidak stabil karena masalah dalam penyiapan pelatihan jaringan Machine learning. Beberapa tindakan umum yang dapat menyebabkan kelemahan seperti ini melibatkan pelatihan dengan data yang terlalu sedikit dan iterasi yang terlalu sedikit atau penggunaan dimensi jaringan yang buruk yang tidak cocok untuk tugas tersebut. Jika pengaturan jaringan pada pengujian salah dan memberikan hasil yang sangat buruk dari waktu ke waktu, selanjutnya dapat memasukkan perbandingan nilai lapisan setelah satu langkah pelatihan (Roberts, (2017). Ini dapat membantu mengetahui tentang adalah kesalahan dalam penyiapan, karena nilai lapisan yang mencapai tensor lain di luar fungsi dapat diverifikasi. Tes Machine learning tidak boleh terlalu lama. Tidak perlu melatih konvergensi dan memeriksa dengan set validasi (Roberts, (2017). Tahun lalu framework mltest6 dirilis, yang menyederhanakan pengujian unit untuk tensorflow. Sebuah port bernama torchtest7 juga telah dirilis yang bekerja dengan pytorch. Secara umum, seperti halnya flakiness karena masalah presisi, disarankan untuk ekstra hati-hati saat merancang pengujian untuk Machine learning. Sebaiknya semua tes harus dijalankan dan diverifikasi beberapa kali secara lokal sebelum check in, karena sifat pelatihan yang terkadang tidak deterministik.

p-ISSN: 2808-8751 e-ISSN : *2798-2513*

```
def test_random_normal(self):
    mean = 0.
    std = 1.
    for k in BACKENDS:
        rand = k.eval(k.random_normal((300, 100 200), mean=mean, stddev=std))
        assert rand.shape == (300, 100 200)
        assert np.abs(np.mean(rand) - mean) < 0.015
        assert np.abs(np.std(rand) - std) < 0.015
```

Gambar 8 Pengujian flakiness dengan masalah pelatihan, dari proyek Keras

Randomness

7 dari 113 (6%) komit berasal dari kategori keacakan. Ini menyiratkan bahwa tes di komit tidak stabil karena menggunakan generator angka acak dengan cara yang tidak berfungsi untuk tes. Jika kegagalan disebabkan oleh angka acak, sangat penting untuk dapat membuat ulang kegagalan tersebut. Menggunakan benih yang ditentukan pengguna membuat ini menjadi kemungkinan. Dengan mampu membuat ulang kegagalan, dimungkinkan untuk men-debug dan memahami dari mana asal masalahnya. Itu bisa dalam pengujian itu sendiri, atau masalah dalam kode yang diuji.

IO

Sejumlah 7 dari 113 (6%) komit berasal dari kategori IO. Kelemahan ini berasal dari masalah dengan deskriptor file atau sumber daya yang tidak ada. Karena ekstensi file dapat berbeda dari waktu ke waktu, pengujian diubah untuk hanya membandingkan nama dasar tanpa ekstensi file, yang mengurangi flakiness. Secara umum untuk menggunakan file untuk menyimpan data pengujian bukanlah ide yang baik karena akan mendapatkan ketergantungan pada sistem file yang dapat membuat pengujian gagal jika jalur yang diberikan tidak berfungsi seperti yang diharapkan pada sistem operasi yang berbeda misalnya, atau jika tidak memiliki izin untuk menggunakan file tersebut maka tesnya juga akan berjalan lebih lambat dari yang diinginkan, ini karena file IO sangat memakan waktu. Cara yang lebih baik daripada membaca file adalah menyimpan data pengujian dalam variabel di kelas pembantu.

```
def test_complex_override_warning(self):
    """Regression test for #19031"""
    with warnings.catch_warnings(record=True) as w:
        warnings.simplefilter("always")
        with override_settings(TEST_WARN='override'):
            self.assertEqual(settings.TEST_WARN, 'override')

        self.assertEqual(len(w), 1)
        self.assertEqual(w[0].filename, __file__)
        self.assertEqual(os.path.splitext(w[0].filename)[0],
                         os.path.splitext(__file__)[0])
        self.assertEqual(str(w[0].message),
            'Overriding setting TEST_WARN can lead to unexpected behaviour.')
```

Gambar 8 Tes flakiness dengan masalah IO, dari proyek Django

Koleksi Tidak Terurut

Sejumlah 7 dari 113 (6%) komit berasal dari kategori pemesanan. Jika pengurutan elemen dapat menentukan hasil tes tanpa memerlukannya, komit diberi label dengan kategori pengurutan. Contoh dari proyek Django adalah pernyataan yang memeriksa daftar yang dihasilkan terhadap daftar yang diharapkan, lihat gambar 9. Daftar selalu memiliki konten yang sama, tetapi urutan elemen dalam daftar yang dihasilkan terkadang berbeda dari daftar yang diharapkan. Kelemahan ini diselesaikan dengan menyortir kedua daftar sebelum menyatakan bahwa isinya sama. Ketergantungan pada urutan tertentu ketika membandingkan dua koleksi yang tidak terurut dapat menyebabkan flakiness. Untuk mengatasi kelemahan seperti itu, ketergantungan pesanan harus dihilangkan. Itu bisa dilakukan dengan menyortir dua koleksi seperti pada contoh Django yang dijelaskan di atas. Sebagian besar Flaky Tes yang dikategorikan dengan unordered collection menyebabkan pada penelitian ini menggunakan sorting untuk mengatasi flakiness. Cara lain adalah dengan menggunakan Counter dict8 dengan Python untuk membandingkan dua koleksi. Counter dict adalah tipe data kumpulan yang menyimpan elemen sebagai kunci kamus dan jumlahnya sebagai nilai kamus. Ini dapat digunakan untuk membandingkan secara efisien jika dua koleksi yang tidak terurut adalah

p-ISSN: 2808-8751 e-ISSN : *2798-2513*

sama. Counter dict sangat berguna jika pengujian berurusan dengan data dalam jumlah besar karena dibangun dalam waktu O(n) dibandingkan dengan menyortir koleksi, yang membutuhkan waktu O(n log n). Namun elemen koleksi harus hashable untuk digunakan dengan Penghitung.

```
def test_relatedfieldlistfilter_foreignkey(self):
    modeladmin = BookAdmin(Book, site)
    request = self.request_factory.get('/')
    changelist = self.get_changelist(request, Book, modeladmin)
    # Make sure that all users are present in the author's list filter
    filterspec = changelist.get_filters(request)[0][1]
    expected = [(self.alfred.pk, 'alfred'), (self.bob.pk, 'bob'),
      ↪  (self.lisa.pk, 'lisa')]
    self.assertEqual(filterspec.lookup_choices, expected)
    self.assertEqual(sorted(filterspec.lookup_choices), sorted(expected))
    ...
```

Gambar 9 Bagian dari pengujian tidak pasti yang menunjukkan masalah dengan koleksi tidak terurut, dari proyek Django

Waktu

Sejumlah 7 dari 113 (6%) komit berasal dari kategori waktu. Komit dalam kategori ini memiliki kelemahan yang berasal dari masalah terkait waktu. Untuk memastikan konsistensi, nilai waktu statis digunakan alih-alih mendapatkan waktu sistem saat ini. Pemformatan waktu adalah alasan lain yang dapat menyebabkan flakiness. Hal ini terjadi pada pengujian lain dari proyek Pandas, lihat gambar 11. Panda menggunakan pemformatan mereka sendiri yang tidak menggunakan angka nol di belakang. Oleh karena itu mereka tidak dapat membiarkan waktu dari datetime.now() memiliki angka nol di belakang. Ini dicapai dengan mengulang sampai waktu tanpa angka nol ditemukan. Kelemahan waktu seringkali dapat dikurangi dengan menggunakan nilai statis untuk membuat operasi lebih deterministik. Terkadang tes perlu dijalankan pada saat-saat khusus. Untuk kasus ini perpustakaan seperti freezegun9 dapat digunakan. Itu mengubah perilaku datetime, memungkinkan untuk "melakukan perjalanan waktu" dalam pengujian.

```
def test_datetime_units(self):
    from pandas.lib import Timestamp
    val = datetime.datetime.now()
    val = datetime.datetime(2013, 8, 17, 21, 17, 12, 215504)
    stamp = Timestamp(val)
    roundtrip = ujson.decode(ujson.encode(val, date_unit='s'))
    self.assert_(roundtrip == stamp.value // 1e9)
    self.assert_(roundtrip == stamp.value // 10**9)
    ...
```

Gambar 10 Bagian dari tes tidak pasti dengan masalah waktu, dari proyek Pandas

p-ISSN: 2808-8751 e-ISSN : *2798-2513*

```
def test_format(self):
    self._check_method_works(Index.format)
    index = Index([datetime.now()])
    # GH 14626
    # our formatting is different by definition when we have
    # ms vs us precision (e.g. trailing zeros);
    # so don't compare this case
    def datetime_now_without_trailing_zeros():
        now = datetime.now()

        while str(now).endswith("000"):
            now = datetime.now()

        return now

    index = Index([datetime_now_without_trailing_zeros()])
    ...

    index = Index([1, 2.0 + 3.0j, np.nan])
    formatted = index.format()
    expected = [str(index[0]), str(index[1]), u('NaN')]
    self.assertEqual(formatted, expected)
    ...
```

Gambar 11: Bagian dari pengujian tidak pasti dengan masalah pemformatan waktu, dari proyek Pandas

Platform

Sejumlah 6 dari 113 (5%) komitmen berasal dari kategori platform. Jika tes tidak stabil karena sesuatu yang terkait dengan platform yang dijalankannya, kategori ini akan digunakan. Serangkaian pengujian yang melibatkan Twisted10 dari proyek Tornado akan berhasil jika diinterpretasikan pada runtime CPython tetapi tidak pada runtime PyPy. Semua pengujian dalam penelitian ini yang tidak stabil karena masalah platform dinonaktifkan seluruhnya atau sebagian. Sehingga, tesdinonaktifkan untuk platform yang ditentukan, karena tidak akan gagal hasil rangkaian tes. Jika tidak ada yang dapat dilakukan untuk mengurangi flakiness platform, menonaktifkan pengujian mungkin satu-satunya pilihan untuk menjaga hasil rangkaian pengujian tetap bersih. Tetapi melewatkan terlalu banyak tes dapat menyebabkan kengerian diam-diam dalam kode produksi (Cunningham, (2006).

```
install:
    ...
    #Twisted runs on 2.x and 3.3+, but is flaky on pypy.
    - if [[ $TRAVIS_PYTHON_VERSION != '3.2' && $TRAVIS_PYTHON_VERSION !=
      ↪  'pypy' && $DEPS == true ]]; then pip install Twisted; fi
    ...
script:
    ...
    - if [[ $TRAVIS_PYTHON_VERSION != '3.2' && $TRAVIS_PYTHON_VERSION !=
      ↪  'pypy' && $DEPS == true ]]; then python $TARGET
      ↪  --ioloop=tornado.platform.twisted.TwistedIOLoop; fi
    ...
```

Gambar 12: File konfigurasi Travis dari proyek Tornado. Tes yang melibatkan Twisted tidak stabil pada runtime PyPy. Flakiness dihentikan dengan membatalkan instalasi Twisted untuk runtime PyPy.

Konkurensi

Sejumlah 5 dari 113 (4%) komit berasal dari kategori konkurensi. Kondisi balapan dan kebuntuan adalah alasan mengapa beberapa komitmen dikategorikan dengan kategori ini. Proyek Certbot memanifestasikan masalah kondisi balapan dengan skrip startup, terlihat pada gambar 13. Mereka memiliki skrip yang seharusnya memulai layanan (Boulder WFE) sebelum uji integrasi boulder-integration.sh dijalankan. Terkadang servis tidak dimulai sebelum pengujian dijalankan, yang menyebabkan kondisi balapan. Flakiness diatasi dengan menambahkan baris 2 dan bagian akhir baris 3 pada gambar 13. Perubahan ini memastikan bahwa skrip startup dijalankan sebelum pengujian. Kelemahan konkurensi seringkali paling baik dimitigasi dengan menambahkan kunci untuk menjamin eksklusi timbal balik ke sumber daya yang

p-ISSN: 2808-8751 e-ISSN : *2798-2513*

dimaksud. Cara lain untuk memperbaiki masalah konkurensi adalah dengan memastikan bahwa operasi dijalankan dalam urutan deterministik, seperti contoh Certbot yang dijelaskan di atas

```
1 install: "travis_retry pip install tox coveralls"
2 before_script: '[ "${TOXENV:0:2}" != "py" ] || ./tests/boulder-start.sh'
3 script: 'travis_retry tox && ([ "${TOXENV:0:2}" != "py" ] || (source
    ↪  .tox/$TOXENV/bin/activate && ./tests/boulder-integration.sh))'
4 after_success: '[ "$TOXENV" == "cover" ] && coveralls'
```

Gambar 13 File konfigurasi Travis dari proyek Certbot. Tes integrasi batu terkelupas karena kondisi balapan dengan dimulainya batu besar. Baris 2 dan bagian terakhir dari baris 3 mengatasi flakiness.

Uji Ketergantungan Urutan

Sejumlah 4 dari 113 (4%) komit berasal dari kategori ketergantungan urutan pengujian. Komit dikategorikan ke dalam kategori ini jika hasil tes tergantung pada urutan tes dijalankan. Masalah ini terjadi saat pengujian bergantung pada status bersama. Misalnya, pengembang mengharapkan satu pengujian yang menyiapkan database untuk dijalankan sebelum pengujian lain yang akan menggunakan database. Jika pengujian tidak dijalankan sesuai urutan yang diinginkan pengembang, hasilnya tidak pasti. Pada gambar 14 dapat dilihat pengujian dari proyek Pandas. Tes ini tidak stabil karena nama tabel yang sama digunakan oleh tes lain. Jika tabel dari pengujian lain masih di-cache, pengujian ini akan gagal. Kelemahan itu diatasi dengan mengubah destination_table menjadi nama tabel yang unik. Sayangnya dalam kasus ini, pengembang tidak memiliki kemungkinan untuk sepenuhnya membersihkan status pengujian mereka karena perubahan pada layanan pihak ketiga (Google BigQuery11) yang berinteraksi dengan mereka terlalu lambat. Cara terbaik untuk mengurangi kelemahan karena ketergantungan urutan pengujian adalah dengan menjaga agar semua pengujian benar-benar terisolasi satu sama lain Martin, (2011). Setiap pengujian sebaiknya dapat mengatur dan membersihkan kondisi yang diperlukan sendiri. Untuk pengujian unit, pendekatan ini bekerja dengan baik, tetapi mungkin lebih bermasalah untuk pengujian fungsional yang lebih besar Martin, (2011). Untuk pengujian yang perlu membangun database besar, seseorang dapat membangunnya sebelum semua pengujian dijalankan dan kemudian melakukan setiap pengujian dalam transaksi. Transaksi dapat dibersihkan dengan melakukan rollback di akhir pengujian, membiarkan status database tidak berubah setelah pengujian selesai. Lupa menutup file yang digunakan oleh beberapa pengujian adalah masalah ketergantungan urutan pengujian lainnya yang dapat menyebabkan kelemahan. Menggunakan pernyataan Python with (Rossum & Coghlan, (2005) saat mengakses file dapat membantu karena file akan ditutup secara otomatis di akhir pernyataan with. Ini memungkinkan beberapa pengujian berikutnya untuk mengakses file yang sama. Pendekatan ini sebaiknya hanya digunakan jika pengujian benar-benar diperlukan untuk mengakses file yang sama. Jika memungkinkan, tes harus memiliki file terpisah atau sumber daya lainnya.

```
1  def test_upload_data_as_service_account_with_key_path(self):
2      destination_table = DESTINATION_TABLE + 11
3      destination_table = "{0}.{1}".format(DATASET_ID + "2", TABLE_ID + "1")
4      test_size = 10
5      df = make_mixed_dataframe_v2(test_size)
6      gbq.to_gbq(df, destination_table, _get_project_id(), chunksize=10000,
7                 private_key=_get_private_key_path())
8      sleep(30)  # <- Curses Google!!!
9      result = gbq.read_gbq(
10         "SELECT COUNT(*) as NUM_ROWS FROM {0}".format(destination_table),
11         project_id=_get_project_id(),
12         private_key=_get_private_key_path())
13     self.assertEqual(result['NUM_ROWS'][0], test_size)
```

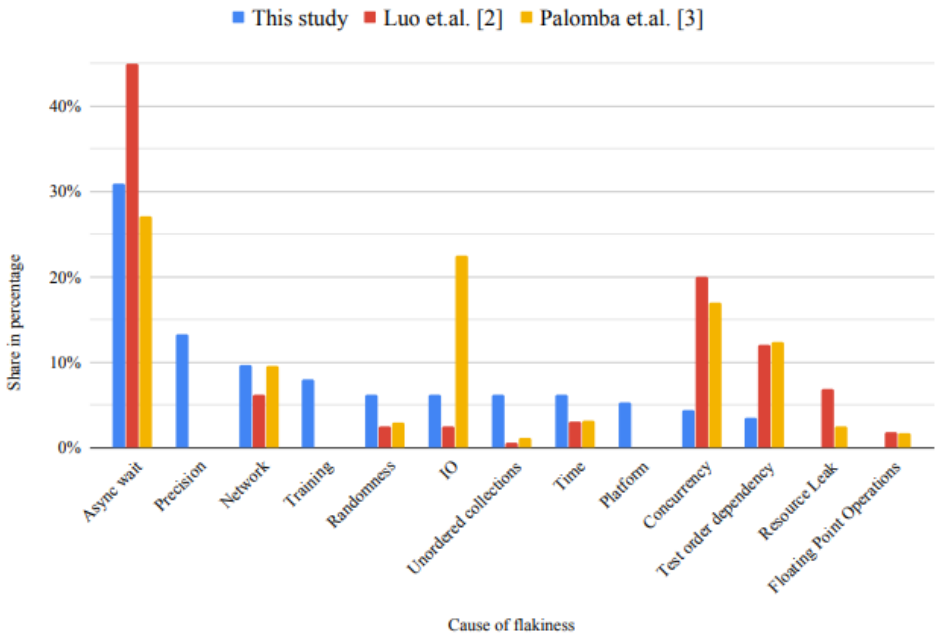
Gambar 14 Tes flakiness dengan masalah ketergantungan pesanan uji, dari proyek Panda

Perbandingan Penyebab Flakiness Python dengan Proyek Java

Studi ini menemukan bahwa masalah flakiness karena menunggu dan jaringan asinkron umum terjadi pada proyek Python dan Java. Masalah flakiness karena IO, konkurensi, dan ketergantungan urutan pengujian cukup umum di proyek Java, tetapi tidak umum di proyek Python menurut hasil penelitian ini. Temuan ini akan membantu peneliti masa depan fokus pada masalah flakiness yang paling penting dalam studi masa depan. Penyebab yang paling universal, terlepas dari teknologi atau bahasa pemrograman, mungkin paling

p-ISSN: 2808-8751 e-ISSN : *2798-2513*

menarik untuk dipelajari di masa depan. Sebuah perbandingan dicapai dengan membandingkan hasil yang ditemukan untuk RQ1 dengan dua penelitian serupa sebelumnya (Luo et al., (2014) dan Palomba & Zaidman, (2018)) tentang kelemahan uji yang terutama meneliti proyek Java. Penelitian Luo et al., (2014) dan Palomba & Zaidman, (2018) dipilih, karena mereka berbagi banyak kategori yang digunakan dalam penelitian ini dan karenanya menjadi perbandingan yang baik dan menarik. Luo et al., (2014) menggunakan metodologi yang mirip dengan penelitian ini dengan inspeksi manual terhadap komitmen. Mereka secara manual mengklasifikasikan 161 komit tes yang rapuh. Tidak semua komit berasal dari proyek khusus Java di (Luo et al., (2014). Tetapi sebagian besar proyek yang diperiksa adalah proyek Java. Palomba & Zaidman, (2018) menggunakan pendekatan yang lebih otomatis dengan menjalankan tes beberapa kali, mengamati perilaku tidak stabil dan kemudian memasangkan kegagalan tes secara manual ke kategori yang ditemukan oleh Luo et al., (2014). Mereka mengidentifikasi dan mengkategorikan 8.829 tes flakiness dengan melakukan ini.



Gambar 15 Perbandingan antara distribusi penyebab flakiness dari penelitian ini dan studi serupa sebelumnya yang terutama berfokus pada proyek di Java

Tabel 3 Perbandingan antara distribusi penyebab flakiness dari penelitian ini dan penelitian serupa sebelumnya yang terutama berfokus pada proyek di Java

Cause of flakiness	This study	Luo et al. 2014 [2]	Palomba et al. 2018 [3]
Async wait	31%	45%	27%
Precision	13%	-	-
Network	10%	6%	10%
Training	8%	-	-
Randomness	6%	2%	3%
IO	6%	2%	22%
Unordered collections	6%	1%	1%
Time	6%	3%	3%
Platform	5%	-	-
Concurrency	4%	20%	17%
Test order dependency	4%	12%	12%
Resource leak	-	7%	3%
Floating point operations	-	2%	2%

p-ISSN: 2808-8751 e-ISSN : *2798-2513*

Hasil dari penelitian ini dan penelitian sebelumnya menunjukkan bahwa pengembang pada umumnya mengalami kesulitan untuk menunggu sumber daya dengan benar saat menulis tes, tidak peduli bahasa apa yang mereka gunakan. Secara umum distribusi penyebab flakiness dari penelitian di Java (Luo et al., (2014) dan Palomba & Zaidman, (2018)) menunjukkan beberapa kategori besar, sedangkan sisanya cukup kecil (≤3%). Studi menemukan bahwa penyebab flakiness pada pengujian Python lebih merata yang dapat dilihat pada gambar 15. Ketergantungan konkurensi dan urutan tes yang keduanya cukup besar dalam studi Java (Luo et al., (2014); Palomba & Zaidman, (2018)) adalah penyebab paling umum dari kelemahan dalam tes Python seperti yang ditemukan oleh penelitian ini. Alasan mengapa konkurensi lebih jarang terjadi dalam proyek Python adalah karena tidak mungkin membuat program multithreaded dengan Python karena Global Interpreter Lock (GIL) Python, (2017).

Masalah jaringan sama umum (10%) dalam proyek Python seperti dalam proyek Java jika dibandingkan dengan karya Palomba et al., (Palomba & Zaidman, (2018). Namun Luo et al., (2014) tidak menemukan tes flakiness karena jaringan umum (6%). Tak satu pun dari dua studi Java (Luo et al., (2014); Palomba & Zaidman, (2018) memiliki tes yang tidak stabil karena pelatihan jaringan Machine learning atau masalah presisi dalam pernyataan. (Luo et al., (2014) dan Palomba & Zaidman, (2018) menggunakan kategori khusus untuk flakiness karena operasi floating point. Dalam tugas akhir ini kesalahan floating point tersebut akan dikategorikan ke dalam kategori presisi. Salah satu alasan mengapa tidak ada penyebab kelemahan terkait Machine learning dalam studi sebelumnya adalah karena sangat sedikit proyek Machine learning yang disertakan dalam kumpulan data mereka. Mahout12 yang merupakan proyek terkait Machine learning diperiksa oleh Luo et al., (2014) tetapi hanya memiliki 2 komit yang cocok dengan kata kunci mereka.

Penyebab Spesifik Bahasa dari Flaky Tes di Proyek Python

Penyebab presisi dan pelatihan flakiness adalah baru dan belum pernah dilaporkan dalam studi tes flakiness sebelumnya. Bagi para praktisi, sangat penting untuk mengetahui penyebab dan cara menanganinya hal tersebut. Terutama mereka yang menulis tes untuk kode Machine learning, karena dua penyebab kelemahan tersebut sebagian besar ditemukan bersamaan dengan tes Machine learning. Pelatihan adalah penyebab paling umum keempat dari kelemahan dengan 9 komitmen, dengan 2 komit berasal dari model Tensorflow dan 7 komit dari Keras. Kategori pelatihan hanya berlaku untuk aplikasi yang menggunakan jaringan Machine learning yang perlu dilatih. Kategori pelatihan ditugaskan untuk pengujian yang tidak stabil, karena pelatihan jaringan Machine learning tidak berfungsi seperti yang diharapkan. Presisi adalah penyebab paling umum kedua dari flakiness dengan 15 kesalahan, dengam 1 komit masing-masing berasal dari model Tensorflow dan Panda, lalu 13 komitmen sisanya berasal dari Keras. Kategori presisi digunakan jika tujuan asersi terlalu sempit, rendah atau tinggi untuk dicapai. Dilihat dari proyek mana yang berkontribusi pada kategori presisi, kita dapat melihat bahwa 14 dari 15 komitmen berasal dari proyek Machine learning, ini menjadikannya penyebab yang cukup spesifik domain. Terungkapnya dua domain spesifik menyebabkan pelatihan dan presisi adalah hasil baru dari penelitian ini. Temuan menunjukkan dua domain spesifik penting untuk dipertimbangkan saat menulis tes aplikasi Machine learning. Kedua kategori tersebut menyumbang sekitar 20% (21 dari 113) dari tes tidak pasti yang ditemukan dalam penelitian ini.

Python adalah bahasa pemrograman yang sangat populer digunakan untuk aplikasi Machine learning (Jetbrains, (2018); Robinson, (2017); Voskoglou, (2017)). Ini adalah alasan terbesar proyek Machine learning memiliki banyak repositori Python populer di GitHub. Saat melatih jaringan atau algoritme Machine learning, hasilnya bisa sepuluh non-deterministik. Sifat domain yang tidak deterministik ini menyebabkan kesulitan saat menulis tes di domain tersebut. Kita harus sangat yakin bahwa akurasi tertentu akan tercapai atau menyetelnya sedikit lebih rendah dari yang diharapkan untuk menghindari flakiness. Non-determinisme pelatihan ini adalah salah satu alasan mengapa kategori presisi dan pelatihan begitu menonjol dalam hasilnya. Kategori platform juga unik dan tidak digunakan sebagai kategori eksplisit dalam studi sebelumnya, tetapi dipertimbangkan dengan cara lain. Luo et al., (2014) misalnya menemukan bahwa 7 dari 161 dari flaky commit dalam kumpulan datanya hanya dapat direproduksi pada platform tertentu. Salah satu alasan mengapa ada beberapa masalah platform yang ditemukan dalam pengujian di Python adalah karena banyak proyek perlu didukung dan diuji di Python 2 dan Python 3. Karena Python 2.7 tidak akan digunakan lagi pada tahun 2020 masalah ini mungkin akan semakin jarang terjadi di masa mendatang. Alasan lain adalah ada beberapa implementasi Python yang tersedia (selain dari implementasi referensi CPython) seperti PyPy13, Jython14 dan Brython15 yang ingin diuji oleh beberapa proyek terhadap kode mereka.

DISKUSI

Ancaman terhadap Validitas

p-ISSN: 2808-8751 e-ISSN : *2798-2513*

Bias Data

Penyebab kelemahan CPython yang paling umum adalah jaringan. Sejumlah 8 dari 11 kejadian flakiness jaringan berasal dari CPython yang berarti jika dihapus, penyebab jaringan turun dari 10% menjadi 3% pada hasil akhir. Karena kategori jaringan adalah yang terbesar ketiga dalam hasil akhir, ini berarti proyek CPython sendiri memengaruhi hasil akhir dengan cara yang cukup signifikan. Penyebab flakiness keras yang paling umum adalah terkait presisi. 13 dari 15 kejadian flakiness presisi ditemukan pada komit Keras yang hampir menghilangkan kategori jika Keras akan dihapus dari kumpulan data. Jika Keras dihilangkan, kategori presisi akan berubah dari kategori flakiness terbesar kedua dengan 13% pada hasil akhir menjadi yang terkecil dengan hanya 2%. Kategori lain yang akan sangat terpengaruh adalah kategori pelatihan komitmen yang berasal dari proyek Keras. Proyek Keras sendiri mempengaruhi hasil akhir dan kesimpulan penelitian dengan cara yang cukup signifikan. Proyek ini juga merupakan bagian yang cukup besar dari keseluruhan kumpulan data komit yang relevan (23 dari 113 komit atau 20%). Fakta meresahkan lainnya yang dapat memengaruhi validitas penelitian adalah sebagian besar data hanya berasal dari dua proyek; Keras dan CPython. Kedua proyek ini menyumbang 49 dari 113 komitmen yang relevan (43%). Dengan menggunakan kata kunci pencarian tambahan untuk menemukan lebih banyak komitmen, pendekatan sampel dapat digunakan. Lebih dari 40 proyek juga dapat diperiksa untuk mencapai keragaman yang lebih besar yang akan mengurangi bias lebih lanjut.

Komitmen Tidak Dikenal

Komit yang tidak diketahui akan sulit untuk diklasifikasi dan ini akan menimbulkan masalah karena data penting bisa saja hilang jika tidak dicantumkan dalam hasil. Sejumlah 25 dari 197 (13%) komitmen yang dianalisis ditandai sebagai tidak diketahui. Thorve et al., (2018) menemukan 11 dari 77 (14%) komit terlalu sulit untuk diklasifikasikan dalam studi mereka tentang tes tidak stabil di aplikasi Android. Luo et al., (2014) menyimpulkan bahwa 40 dari 201 (20%) komit yang mereka periksa terlalu sulit untuk diklasifikasi dalam studi mereka tentang tes flakiness dalam proyek Apache. Karena penelitian ini telah menggunakan metodologi yang mirip dengan Luo et al., (2014) dan Thorve et al., (2018), komitmen yang tidak diketahui tampaknya berada pada tingkat yang dapat diterima dan seharusnya tidak menjadi masalah validitas penelitian

Inspeksi Manual

Melakukan pelabelan secara manual dapat menyebabkan hasil yang salah. Untuk meminimalkan kemungkinan kesalahan, setiap komit diperiksa secara menyeluruh dan jika diperlukan penulis akan berkonsultasi dengan orang berpengetahuan lainnya untuk mendapatkan pendapat kedua.

Pemilihan Proyek

Untuk mendapatkan pilihan proyek yang berbeda dalam penelitian, akan menarik untuk menggunakan sesuatu yang lain selain bintang GitHub. SourceRank menggunakan kombinasi metrik komunitas, distribusi, dokumentasi, dan penggunaan. Ini memperhitungkan tingkat penggunaan kembali proyek. Fakta bahwa hanya 15 dari 40 proyek Python berbintang teratas yang memiliki pesan komit tentang pengujian yang tidak stabil mungkin dapat dikaitkan dengan fakta bahwa cukup banyak dari 40 repositori teratas pada saat itu tidak berisi proyek kode sumber terbuka "biasa".

KESIMPULAN

Hasil penelitian ini memperkuat klaim bahwa menunggu asinkron adalah alasan paling umum untuk pengujian yang tidak stabil. Semua studi kelemahan serupa sebelumnya (Qingzhou Luo et al., (2014); Palomba & Zaidman, (2018); Vahabzadeh e al., (2015); Thorve et al., (2018) yang terutama berfokus pada proyek Java atau Android telah mencapai kesimpulan yang sama. Namun penelitian ini sampai pada kesimpulan yang sangat berbeda mengenai banyak penyebab lain dari flakiness. Ketergantungan konkurensi dan urutan pengujian yang merupakan salah satu penyebab paling umum dari kelemahan (Qingzhou Luo et al., (2014); Palomba & Zaidman) adalah yang paling tidak umum dalam proyek Python. IO yang merupakan penyebab paling umum kedua dari flakiness menurut Palomba & Zaidman, (2018) hampir tidak lazim dalam proyek Python. Studi ini juga menunjukkan bahwa konkurensi bukanlah penyebab umum kelemahan dalam proyek Python dibandingkan dengan proyek Java. Kemungkinan alasannya mungkin ada lebih sedikit kode ganda dalam proyek Python, ini karena kita tidak dapat membuat program multithreaded dengan Python yang memanfaatkan sepenuhnya sistem multiprosesor karena GIL (Global Interpreter Lock) (Python, (2017). Bahkan jika program Python diimplementasikan untuk

p-ISSN: 2808-8751 e-ISSN : *2798-2513*

dijalankan di banyak utas, sebagian besar jenis utas tidak akan pernah dieksekusi pada waktu yang bersamaan. Penyebab paling umum keempat dari kelemahan pelatihan hanya berlaku untuk proyek yang menggunakan komponen Machine learning yang perlu dilatih. Penyebab paling umum kedua, presisi juga berlaku pada sebagian besar tes Machine learning. Proyek Machine learning (Keras, model Tensorflow, dan Scikit-learn) menyumbang 30 dari 113 (27%) komitmen dalam kumpulan data total. Sejumlah 21 dari 30 komitmen tersebut menunjukkan kelemahan karena penyebab khusus domain yang terkait dengan tes Machine learning. Ini menunjukkan bahwa proyek Machine learning memiliki sebagian besar pengujian khusus dengan penyebab kelemahan khusus domainnya sendiri.

Pekerjaan masa depan

Untuk mengurangi ancaman validitas, inspeksi manual mungkin dapat dibuktikan. Banyak orang dapat meninjau dan mengkategorikan setiap komit secara terpisah. Jika komit akan ditandai dengan kategori yang berbeda oleh orang yang memeriksanya, mereka dapat mendiskusikan masalah tersebut bersama-sama dan klasifikasinya mungkin akan lebih sedikit rawan kesalahan. Terdapat kemungkinan bias dari beberapa proyek yang menyumbangkan sebagian besar kumpulan data adalah masalah yang dapat diatasi di masa mendatang dengan menggunakan kumpulan data awal yang lebih besar dan komitmen sampel dari proyek.

DAFTAR PUSTAKA

1. Akli, A., Haben, G., Habchi, S., Papadakis, M., & Traon, Y. L. (2022). Predicting Flaky Tests Categories using Few-Shot Learning. arXiv preprint arXiv:2208.14799.
2. Anirban. Basu. Software quality assurance, testing and metrics. Prentice Hall Of India, 2015. isbn: 9788120350687. url: https://books.google.de/books?id=aNTiCQAAQBAJ&pg=PA150#v=onepage&q&f=false.
3. Apache Software Foundation. Apache Projects Directory. url: https://projects.apache.org/.
4. Apache Software Foundation. Apache Software Foundation. url: https://www.apache.org/.
5. Arash Vahabzadeh, Amin Milani Fard, and Ali Mesbah. “An empirical study of bugs in test code”. In: 2015 IEEE 31st International Conference on Software Maintenance and Evolution, ICSME 2015 - Proceedings (2015), pp. 101–110. issn: 15461718. doi: 10.1109/ICSM. 2015.7332456.
6. Azeem Ahmad, Ola Leifler, and Kristian Sandahl. “Empirical Analysis of Factors and their Effect on Test Flakiness - Practitioners’ Perceptions”. In: (June 2019). url: http://arxiv.org/abs/1906.00673.
7. Chase Roberts. How to unit test machine learning code. 2017. url: https://medium.com/@keeper6928/how-to–unit-test-machine-learning-code-57cf6fd81765.
8. Christina Voskoglou. What is the best programming language for Machine Learning? 2017. url: https://towardsdatascience.com/what-is-the-best-programming-language-formachine-learning-a745c156d6b7.
9. D. Thomas and A. Hunt. “Mock objects”. In: IEEE Software 19.3 (May 2002), pp. 22–24. issn: 0740-7459. doi:10.1109/MS.2002.1003449. url: http://ieeexplore.ieee.org/document/1003449/.
10. David Robinson. Why is Python Growing So Quickly? - Stack Overflow Blog. 2017. url: https://stackoverflow.blog/2017/09/14/python-growing-quickly/.
11. Digital Ocean. Currents Q1 2018 - A quarterly report on developer trends in the cloud. Tech. rep. 2018. url: https://assets.digitalocean.com/currents-report/DigitalOcean-Currents-Q1-2018.pdf.
12. Emelie Engström, Per Runeson, and Mats Skoglund. “A systematic review on regression test selection techniques”. In: (2009). doi:10.1016/j.infsof.2009.07.001. url: www.theiet.org/publishing/inspec.
13. Fabio Palomba and Andy Zaidman. “Does refactoring of test smells induce fixing flaky tests?” In: Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017 (2017), pp. 1–12. doi: 10.1109/ICSME.2017.12.
14. Fabio Palomba and Andy Zaidman. The Smell of Fear: On the Relation between Test Smells and Flaky Tests - Online Appendix. 2018. doi:10.1098/rspb.2007.1009. url:https://dibt.unimol.it/staff/fpalomba/documents/J18.pdf%20https://tinyurl.com/ycnmnd6w.
15. GitHub. About stars. url: https://help.github.com/articles/about-stars/.

p-ISSN: 2808-8751 e-ISSN : *2798-2513*

16. GitHub. GitHub. url: https://github.com/.
17. GitHub. Search API. url: https://developer.github.com/v3/search/.
18. Guido van Rossum and Nick Coghlan. PEP 343 – The with Statement. 2005. url: https://www.python.org/dev/peps/pep0343/.
19. Haben, G., Habchi, S., Papadakis, M., Cordy, M., & Traon, Y. L. (2023). The Importance of Discerning Flaky from Fault-triggering Test Failures: A Case Study on the Chromium CI. arXiv preprint arXiv:2302.10594.
20. Jetbrains. Python Developers Survey 2018 Results. 2018. url: https://www.jetbrains.com/research/python-developers-survey-2018/#types-of-development.
21. Jonathan Bell et al., "DeFlaker: Automatically Detecting Flaky Tests". In: Icse 2018 (2018), pp. 433–444. doi: 10.1145/3180155.318016
22. Keenan Szulik. Don't judge a project by its GitHub stars alone. 2017. url: https://blog.tidelift.com/dont-judge-a-project-by-its-github-stars-alone
23. Lam, W., Winter, S., Wei, A., Xie, T., Marinov, D., & Bell, J. (2020). A large-scale longitudinal study of flaky tests. Proceedings of the ACM on Programming Languages, 4(OOPSLA), 1–29. https://doi.org/10.1145/3428270
24. Libraries.io. Overview - Sourcerank. url: https://docs.libraries.io/overview.html#sourcerank.
25. Martin Fowler. ContractTest. 2011. url: https://martinfowler.com/bliki/ContractTest.html.
26. Martin Fowler. Eradicating Non-Determinism in Tests. 2011. url: https://martinfowler.com/articles/nonDeterminism.html.
27. Martin Fowler. Refactoring: improving the design of existing code. AddisonWesley Professional, 2018.
28. Michael Hilton et al., "Trade-offs in continuous integration: assurance, security, and flexibility". In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017. New York, New York, USA: ACM Press, 2017, pp. 197–207. isbn: 9781450351058. doi: 10.1145/3106237.3106270. url: http://dl.acm.org/citation.cfm?doid=3106237.3106270.
29. Microsoft. Eliminating Flaky Tests - Azure DevOps. 2017. url: https://docs.microsoft.com/en-us/azure/devops/learn/devops-at-microsoft/eliminating-flaky-tests
30. Muhammad Khatibsyarbini et al., Test case prioritization approaches in regression testing: A systematic literature review. Jan. 2018. doi:10.1016/j.infsof.2017.08.014. url: https://www.sciencedirect.com/science/article/abs/pii/S0950584916304888.
31. pytest documentation. Skip and xfail: dealing with tests that cannot succeed. url: https://docs.pytest.org/en/latest/skipping.html.
32. Python Wiki. Python2orPython3. url: https://wiki.python.org/moin/Python2orPython3.
33. Python. GlobalInterpreterLock - Python Wiki. 2017. url: https://wiki.python.org/moin/GlobalInterpreterLock.
34. Python. PEP 373. url: https://www.python.org/dev/peps/pep-0373/.
35. Qi Luo et al., How Do Static and Dynamic Test Case Prioritization Techniques Perform on Modern Software Systems? An Extensive Study on GitHub Projects. 2018. doi: 10.1109/TSE.2018.2822270. url: http://ieeexplore.ieee.org/document/8329518/.
36. Qin, Yihao. (2022). Online Repository for ICSME\'2022 Paper "PEELER: Learning to Effectively Predict Flakiness without Running Tests". Zenodo. https://doi.org/10.5281/zenodo.7066904
37. Qingzhou Luo et al., "An empirical analysis of flaky tests". In: Proceedings of the 22nd AC M SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014 (2014), pp. 643–653. issn: 1600-6135. doi: 10.1145/2635868.2635920. url: http://dl.acm.org/citation.cfm?doid=2635868.2635920.

p-ISSN: 2808-8751 e-ISSN : *2798-2513*

38. Sai Zhang et al., “Empirically Revisiting the Test Independence Assumption”. In: (2014). doi: 10.1145/2610384.2610404. url: http://dx.doi.org/10.1145/2610384.2610404.
39. Stack Overflow Developer Survey. 2019. url: https://insights.stackoverflow.com/survey/2019#most-popular-technologies.
40. Swapna Thorve, Chandani Sreshtha, and Na Meng. “An Empirical Study of Flaky Tests in Android Apps”. In: Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, Sept. 2018, pp. 534–538. isbn: 978-1-5386-7870-1. doi:10.1109/ICSME.2018.00062. url: https://ieeexplore.ieee.org/document/8530060/.
41. Takfarinas Saber et al., “A Hybrid Algorithm for Multi-Objective Test Case Selection”. In: 2018 IEEE Congress on Evolutionary Computation, CEC 2018 - Proceedings. IEEE, July 2018, pp. 1–8. isbn: 9781509060177. doi: 10.1109/CEC.2018.8477875. url: https://ieeexplore.ieee.org/document/8477875/.
42. Tariq M. King et al., “Towards a Bayesian Network Model for Predicting Flaky Automated Tests”. In: Proceedings - 2018 IEEE 18th International Conference on Software Quality, Reliability, and Security Companion, QRS-C 2018 (2018), pp. 100–107. doi: 10.1109/QRSC.2018.00031.
43. Travis CI. url: https://travis-ci.org/.
44. Vahid Garousi, Ramazan Özkan, and Aysu Betin-Can. “Multi-objective regression test selection in practice: An empirical study in the defense software industry”. In: Information and Software Technology 103 (Nov. 2018), pp. 40–54. issn: 09505849. doi:10.1016/j.infsof.2018.06.007. url: https://www.sciencedirect.com/science/article/abs/pii/S0950584918301186.
45. Ward Cunningham. Bugs In The Tests. 2006. url: http://wiki.c2.com/?BugsInTheTests.
46. Wing Lam et al., iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. Tech. rep. University of Illinois, 2019. url: http://mir.cs.illinois.edu/marinov/publications/LamETAL19iDFlakies.pdf.
47. Zhang, L., Radnejad, M., & Miranskyy, A. (2023). Identifying Flakiness in Quantum Programs. arXiv preprint arXiv:2302.03256.