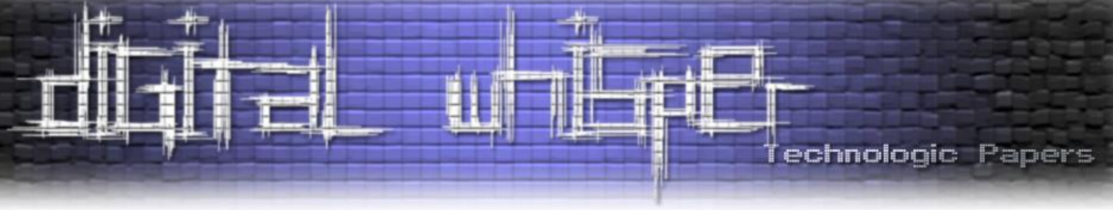


Digital Whisper

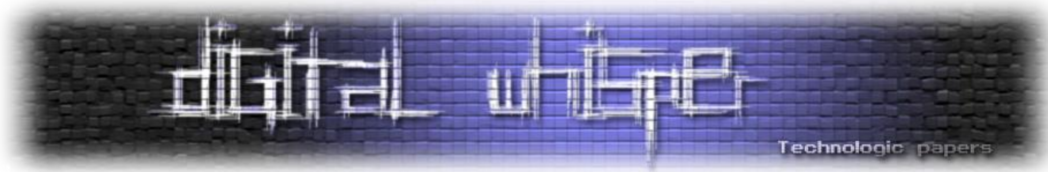
גליון 104, מרץ 2019

מערכת המגזין:

מייסדים:	אפיק קסטיאל, ניר אדר
מוביל הפרויקט:	אפיק קסטיאל
עורכים:	אפיק קסטיאל
כתבים:	דן רווח, ליאור בר-און ועידו קנר

יש לראות בכל האמור במגזין Digital Whisper מידע כללי בלבד. כל פעולה שנעשית על פי המידע והפרטים האמורים במגזין Digital Whisper הינה על אחריות הקורא בלבד. בשום מקרה בעלי Digital Whisper ו/או הכותבים השונים אינם אחראים בשום צורה ואופן לתוצאות השימוש במידע המובא במגזין. עשיית שימוש במידע המובא במגזין הינה על אחריותו של הקורא בלבד.

פניות, תגובות, כתבות וכל הערה אחרת - נא לשלוח אל editor@digitalwhisper.co.il



דבר העורכים

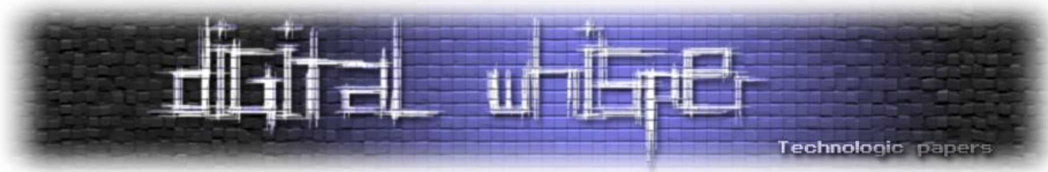
ברוכים הבאים לדברי הפתיחה של הגליון ה-104. הפעם, נחסוך מכם את הגיגינו ונדלג על דברי הפתיחה. הגליון הנ"ל מוקדש כולו לגשושית **בראשית**. ואנו מאחלים לה ולכל צוות SpaceIL בהצלחה ענקית במסע שלה אל עבר הירח!

בהצלחה!

וכמובן, רגע לפני שנשחרר אתכם, ברצוננו להגיד תודה לכל מי שתרם מזמנו החודש, ישב וכתב לכם מאמרים. אז תודה רבה ל**דן רווח**, תודה רבה ל**ליאור בר-און** ותודה רבה ל**עידו קנר**!

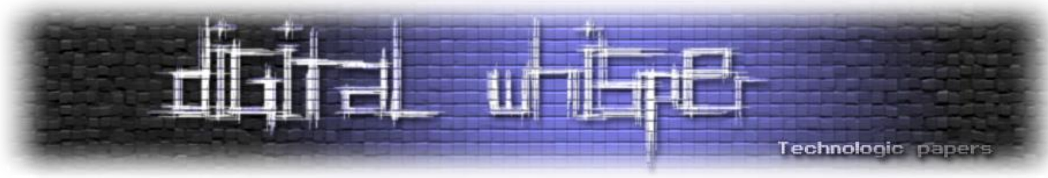
קריאה נעימה,

אפיק קסטיאל וניר אדר



תוכן עניינים

דבר העורכים	2
תוכן עניינים	3
זיהוי אתרי פישינג בעזרת Machine Learning	4
לקבל מושג ירוק על קוברנטיס (Kubernetes)	30
באגים נסתרים	41
דברי סיכום	48



זיהוי אתרי פישינג בעזרת Machine Learning

מאת דן רווח

הקדמה

מאמר זה יעסוק בזיהוי אתרי פישינג בעזרת טכנולוגיה הנקראת למידת מכונה. אנו נתמקד בעיקר באלגוריתמים: Linear Regression, K-Nearest Neighbour, Decision-Tree. וכיצד ניתן להשתמש בהם על מנת להצליח לזהות אתרי פישינג.

נעבור על 2 מחקרים בתחום נראה את הגישה שלהם להתמודדות עם הבעיה, ולאחר מכן ננסה לכתוב קוד שידע לזהות אתרי פישינג בעזרת python ו-turicreate.

בזיהוי אתרי פישינג על-ידי למידת מכונה, אנו נגדיר פרמטרים שונים ונאמן את המודל על מסד נתונים שיכיל פרטים לגבי אתרים חשודים ושאינם חשודים (supervised training), ונלמד את המודל לזהות מבינהם מיהו האתר החשוד, ומי הוא האתר הלגיטימי.

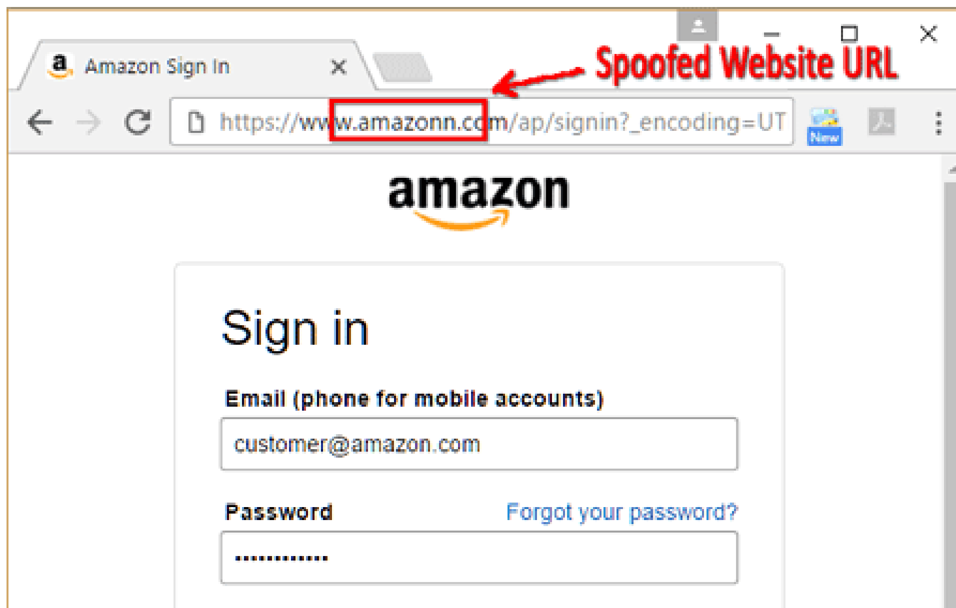
לצורך הדגמת האלגוריתמים, נציג בסוף המאמר הרצה של אלגוריתמי הלמידה על מאגרי מידע של אתרים חשודים.

תיאור הבעיה

בעשורים האחרונים, בעקבות התפתחות עידן האינטרנט, החלו להופיע רמאויות "חדשות". מסוג Phishing (או בעברית: דיוג). זהו ניסיון לגניבת מידע רגיש על-ידי התחזות ברשת האינטרנט. למשל, פישינג מתבצע באמצעות התחזות לגורם לגיטימי המעוניין לקבל את המידע. לרוב, שולח הגורם המתחזה הודעת מסרים מיידית, דואר אלקטרוני או בשם אתר אינטרנט מוכר (בבעיה הזו בעצם נתמקד). האתגר הינו לזהות אתרי Phishing בעזרת Machine Learning ונראה את הטכניקות והפרמטרים (Features) שניתן להוציא על מנת לזהות בצורה אפקטיבית אתר פישינג.

לדוגמא: הרבה רשתות תקיפה רבות מתחילות מלינק לאתר פישינג. אימייל אשר נראה לגיטימי יכול להישלח לעובד אשר לא חושד, וברגע שהוא לוחץ על הלינק כל תהליך התקיפה מתחיל, ואיתו איבוד המידע והנזק. לכן נראה כי יש מוטיבציה רבה לזהות את האתרים הללו בשלבים מוקדמים על מנת להתריע על לינקים לאתרים מזיקים ולמנוע מהעובד כניסה אליהם.

דוגמא לאתר פישינג אשר מנסה להתחזות לאמזון דוט קום, כמובן ברגע שהלקוח יזין את הפרטים שלו, מידע זה יגיע ישירות לתוקף, ובכך יוכל לאבד את פרטי הגישה לחשבון ואף לגרום לו במקרה הנ״ל להפסיד כסף דרכי שירותי Amazon web.



מבחינת פתרונות לזיהוי אתרי פישינג, ישנם שירותים רבים אשר מציעים שירות של ״רשימה-שחורה״ של שירותים או מאגרים אשר מכילים אתרים בעלי פוטנציאל להיות אתרי פישינג. אבל כמו כל פתרון אשר מתבסס על ״חתימה״ נראה כי לא ניתן יהיה לזהות אתרי פישינג חדשים.

המטרה העיקרית של הפתרון תהיה לזהות בצורה חכמה אתר פישינג בעזרת מנגנון של למידת מכונה, אשר ידע לזהות אתרים אלו מראש ולהתריע עליהם.

הבעיה מעניינת מאחר וקיימים פרמטרים רבים באמצעותם ניתן להסיק כי אתר מסויים הוא אתר פישינג, ולכן יש לזהות את הפרמטרים הללו ולדעת כיצד להשתמש בהם על מנת להגיע לתוצאות האופטימליות.

מבוא לאגוריתם הלמידה

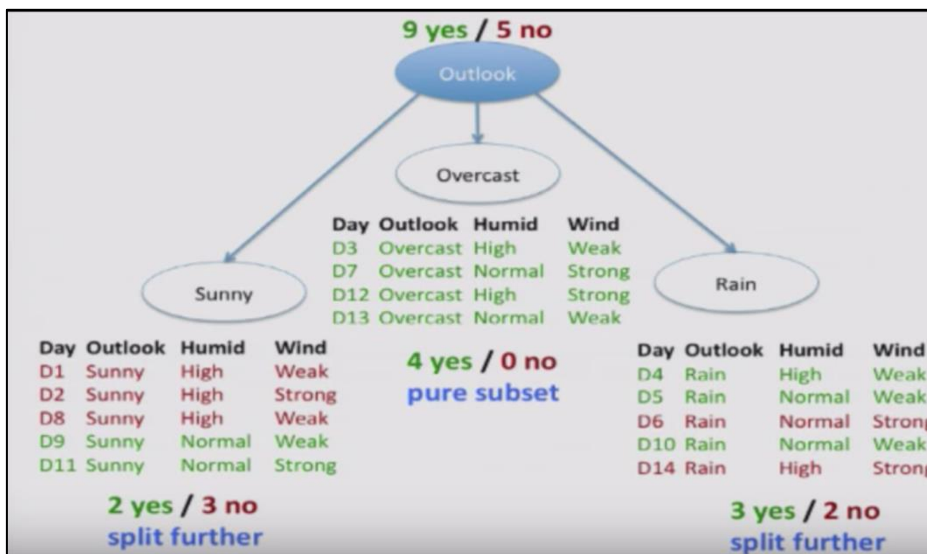
הפרק מתבסס על המאמר Classifcation Approach Learning Machine. במאמר נבחרו 4 אלגוריתמי למידת מכונה אשר נבדקו זה ליד זה על מנת לבצע זיהוי מוצלח של אתרי Phishing.

האלגוריתמים הנבחרים:

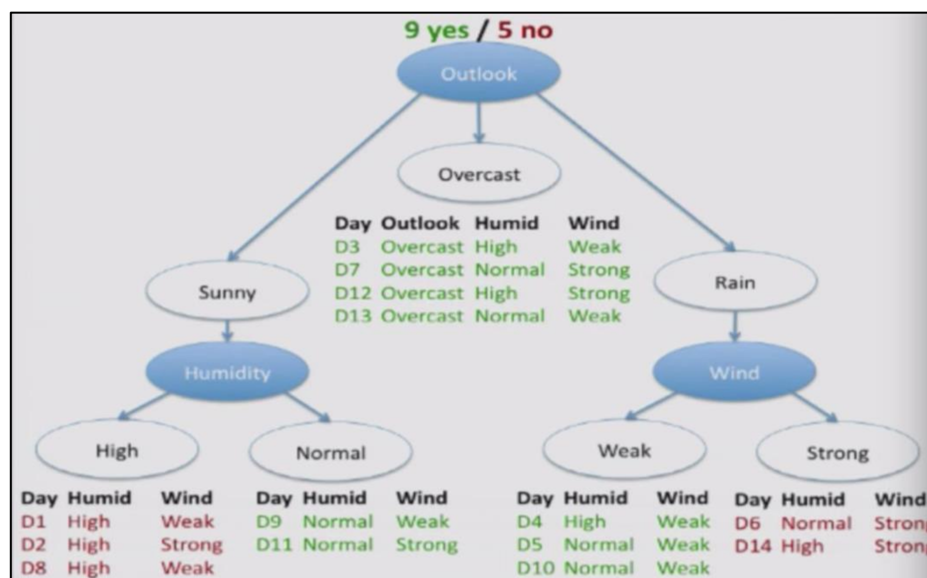
1. עץ בחירה (Decision Tree) - זהו אלגוריתם אשר משמש כמודל חיזוי, אשר ממפה תצפיות על פריט ויוצר מסקנות על ערך היעד שלו. בכללי, עץ החלטה הוא עץ בינארי המורכב מצמתי החלטה שבכל אחד מהם נבדק תנאי מסוים על מאפיין מסוים של התצפיות, ומוגדרים העלים המכילים את הערך החזוי עבור התצפית המתאימה למסלול שמוביל אליהם בעץ.

לדוגמא: אם אחת עמודות הפ'יצרים היא "חיבור SSL" והתשובות האפשריות הם כן ולא. אזי ייבנה עץ החלטה אשר הצומת הראשית שלו תהיה "חיבור SSL" אשר יחובר לשני ענפים - כן ולא, נניח כי כל האתרים במאגר שלהם יש חיבור ssl אינם אתרי פישינג. במקרה הזה הענף של האופציה "כן" יהיה pure subset ושם יסתיים תהליך ה-Split. כלומר ברגע שאתר הוא ssl נוכל לדעת בוודאות כי הוא אינו אתר פישינג (לפי המאגר).

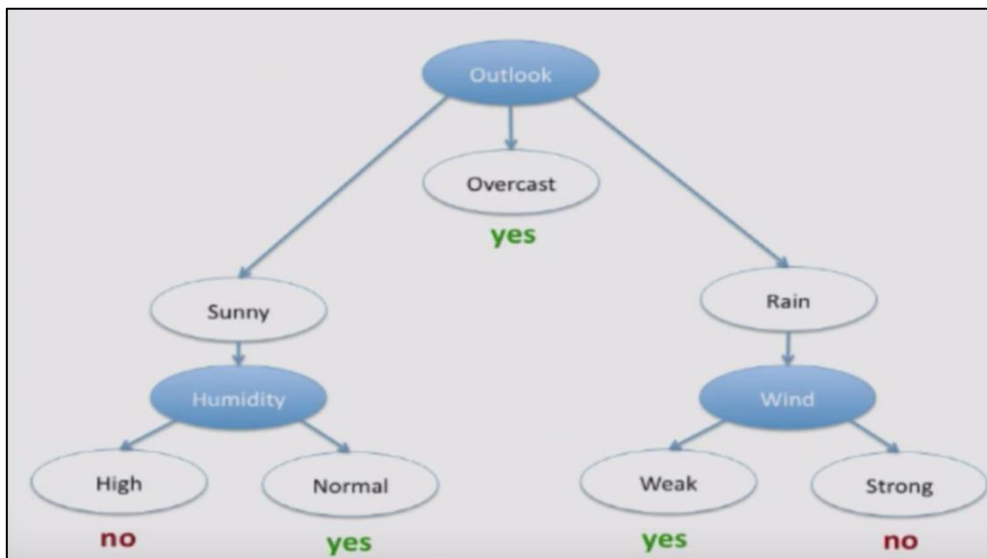
נניח כעת שבאופציה השניה של ה-"לא" יש סיכוי של 60% שהאתר הוא אתר פישינג במקרה הזה נמשיך לחלק את העץ עם עמודה נוספת למשל: "אורך ה-URL" ונמשיך הלאה ככה עד שנגיע ל- pure subsets בכל העלים.



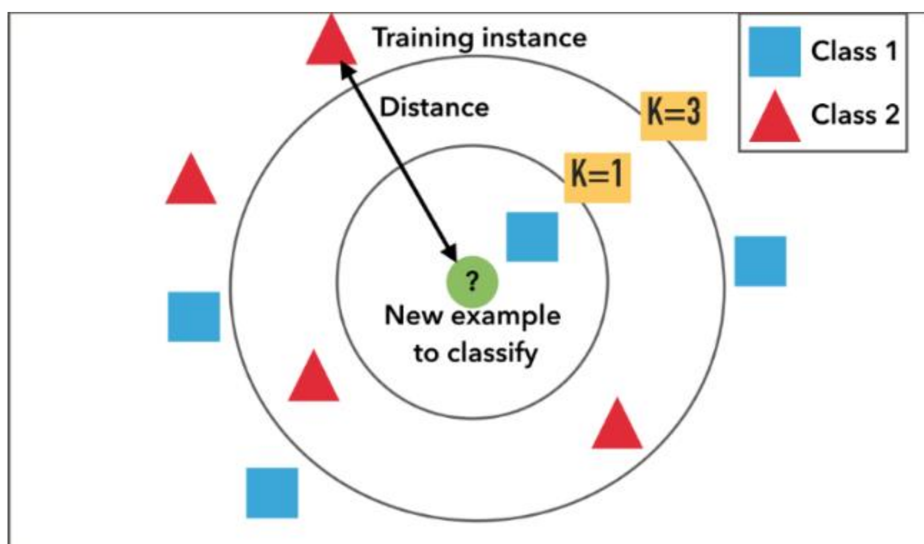
בדוגמא למעלה: האם שחקן מסויים יצא לשחק טניס? רואים שבימים שבהם התחזית היא overcast על פי התצפיות הקודמות של האדם נראה כי הוא ייצא לשחק. בשאר המקרים אין תשובה מוחלטת ולכן תהליך החלוקה ימשיך.



ובסופו של דבר נקבל את עץ ההחלטה הבא:

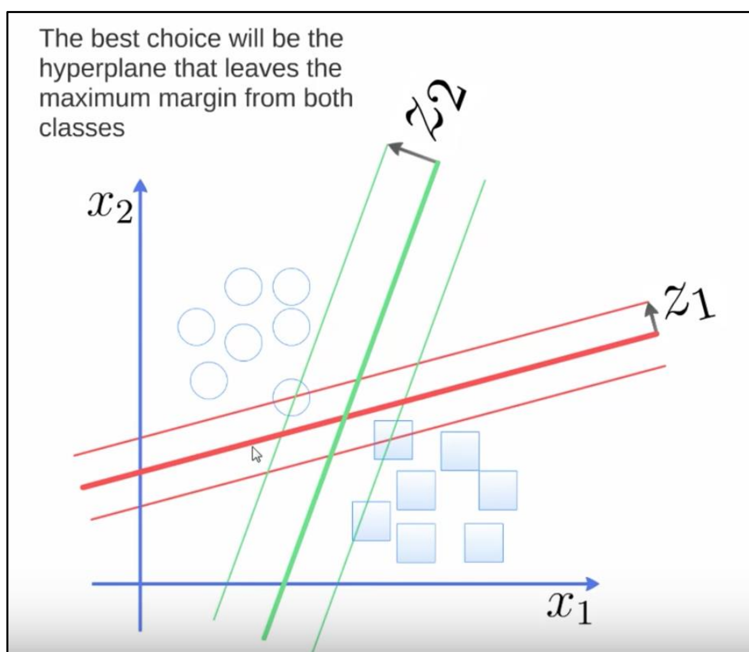


2. אלגוריתם שכן קרוב (K Nearest Neighbours) הינו אלגוריתם חסר פרמטרים לסיווג ולרגרסיה מקומית. בשני המקרים הקלט תלוי ב-k התצפיות הקרובות במרחב התכונות. בהינתן קלט של דוגמה חדשה, האלגוריתם משייך לקבוצה. הדוגמה משויכת למחלקה הנפוצה ביותר בקרב k השכנים הקרובים (כאשר k מוגדר כמספר חיובי שלם, בדרך כלל מספר קטן). אם k=1 האובייקט משויך למחלקה של השכן הבודד הקרוב ביותר.



בדוגמא למעלה: למשל עבור K=1 נקבל כי האיבר החדש הוא מסוג class 1, אבל עבור k=3 נקבל תשובה אחרת שכן האיבר החדש יהיה מסוג class 2.

3. מכונת וקטורים תומכת (Support Vector Machine) - היא טכניקה של למידה מונחית (supervised learning), המשמשת לניתוח נתונים לסיווג ולרגרסיה. דוגמאות האימון מיוצגות כווקטורים במרחב לינארי. עבור בעיות סיווג, בשלב האימון מתאימים מסווג שמפריד נכון ככל האפשר בין דוגמאות אימון חיוביות ושליליות. המסווג שנוצר ב-SVM הוא המפריד הלינארי אשר יוצר מרווח גדול ככל האפשר בינולבין הדוגמאות הקרובות לו ביותר בשתי הקטגוריות.



בדוגמא למעלה: נראה כי z2 משאיר מרווח גדול יותר מ-1z ולכן הוא יהיה המשוואה המפרידה הטובה ביותר אשר תבחר.

4. אלגוריתם רגרסיה לינארית (Linear Regression) - רגרסיה ליניארית היא שיטה מתמטית למציאת הפרמטרים של הקשר בין משתנה בלתי תלוי X למשתנה תלוי Y, בהנחה שהקשר ביניהם ליניארי, כלומר מהצורה Y=aX+b. השיטה משמשת לניתוח מדגמים סטטיסטיים. נוסחת הרגרסיה הליניארית מחשבת את הקו הישר שעובר דרך הנקודות שבמדגם

משמעות הפרמטרים

		Predicted/Classified	
		Negative	**Positive**
Actual	**Negative**	998	0
	Positive	1	1

על מנת להסביר את משמעות הפרמטרים נציג לפני כן את ה-Confusion Matrix

הטבלה מציגה את התוצאות האמיתיות ואת אלה שנוחשו ע״י האלגוריתם. ניתן לראות שהטבלה מציגה אלגוריתם אשר מציג Accuracy של 99.9%.

הנקודה הבודדת שבה האלגוריתם טועה הוא בחישוב של Positive בודד כ-Negative. עפ״י תלות בבעיה שבה אנו מנסים לפתור זו יכולה להיות טעות הניתנת לסבילה. אבל אם נניח אנו מנסים לאתר טרוריסט והאלגוריתם מזהה אותו כאחד שאינו טרוריסט, זוהי טעות הרבה פחות זניחה ולכן ישנם כמה metrics שעל פיהם ניתן למדוד את איכות התוצאה.

Precision & Recall נמדדים ע״י הנוסחאות הבאות:

$$\text{Precision} = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

$$\text{Recall} = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

כאשר הערכים שמציבים בהם, הם הערכים אשר מגיעים מטבלת ה-Confusion:

		Predicted	
		Negative	**Positive**
Actual	**Negative**	True Negative	False Positive
	Positive	False Negative	True Positive

ניתן לראות שהמכנה המשותף בנוסחאת ה-Precision הוא סך כל ה- Predicted Positiveבין אם trueאו false. ולכן, ניתן להסיק כי Precision מדבר על כמה מדוייק המודל על פי הערכים החיוביים, וכמה מהם באמת חיוביים. זהו יחידת מידה טובה כאשר רוצים לבדוק מה עלות ה-false positive ומתי שהעלות הזו יקרה, למשל כאשר מייל לגיטימי יזוהה כמייל ספאם וישלח ל-Junk, זוהי עלות גבוהה של False Positive.

בנוסחאת ה-Recall, הנוסחא מחשבת כמה Actual Positive במודל שלנו סומנו כ-True Positive. ולכן Recall יחידת מידה טובה כאשר יש עלות גבוהה ל-False Negative. למשל, אם האלגוריתם מזהה אנשים אשר נשאי מחלה מסוכנת ככאלו שבריאים לחלוטין זוהי דוגמא לעלות גבוהה.

אלגוריתם ה-F1 Score הוא שילוב של התוצאות שקיבלנו יחדיו לנוסחא הבאה:

$$F1 = 2 \times \frac{Precision*Recall}{Precision+Recall}$$

יחידת המידה הזו הכרחית כאשר אנו רוצים לבדוק את האיזון בין ה-Precision ל-Recall.

תיאור הניסויים ומסקנותיהם

הניסויים שבוצעו חולקו לשלושה שלבים. בשלב הראשון מתבצעת יצירה של מודל עץ החלטות. וביצוע בחירת הפרמטרים בקפידה רבה דרך תהליך מחזורי, בנוסף הם מוצגת טכניקת ה-"קיצוץ" (pruning).

חלק מהפרמטרים נבחרים לפני הבנייה ואימון המודל. ישנם 6 פרמטרים אשר משמשים ליצירת מודל עץ החלטה, ולכל אחד מהפרמטרים יש טווח של ערכים. הרעיון של שימוש ב"קיצוץ" עם טכניקת עץ ההחלטה היא על מנת למנוע התאמת יתר של הבעיה. קיימות שתי גישות על מנת למנוע התאמת יתר, ביצוע הקיצוץ לפני או אחרי. במחקר הנוכחי בוצע רק קיצוץ אחרי. לטענתם הקיצוץ המוקדם יכול לגרום לפספוסים.

התהליך המחזורי הציג פרמטר של "ביטחון", שעליו מסתמך תהליך ה-Feature Extraction. ישנו תהליך שממנו אנו מוציאים Features ונותנים להם משקל על פי פרמטר ה-"ביטחון".

טבלת הפ'יצרים מהמאמר:

Feature	Weight		
	Dataset 1 (50:50)	Dataset 2 (70:30)	Dataset 3 (30:70)
ssl_connection	1.0	1.0	1.0
google_position	0.658	0.797	0.745
alexa_rank	0.542	0.555	0.532
google_page_rank	0.408	0.518	0.369
dots	0.146	0.114	0.288
long_url	0.103	0.096	0.195
at_symbol	0.094	0.080	0.101
frame	0.045	0.062	0.048
ip_address	0.019	0.062	0.047
redirect	0.008	0.059	0.024
hexadecimal	0.004	0.022	0.010
submit	0.000	0.000	0.000

הטבלה מציגה את המשקלים לפי ה-Gain Ratio אשר מחושב ע"י:

$$Gain\ Ratio = \frac{Information\ Gain}{Split\ Information}$$

ה-Information Gain הוא ממדידה של כמה "מידע" פ'יצר נותן לנו.

הוא מחושב ע"י הנוסחא הבאה:

Information gain = entropy (parent) – [weightes average] ★ entropy (children)

ו-Entropy הוא הדרך של עץ ההחלטה להחליט לפצל את המידע, הוא משפיע על עץ ההחלטה ומסמן לו גבולות.

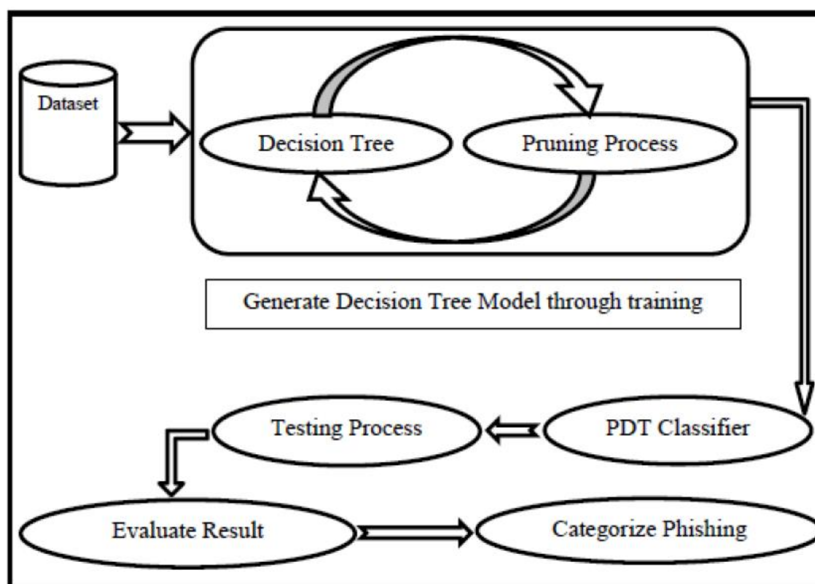
קיים הבדל קטן במשקלים בין כל ה-פ'יצרים בכל שלושת ה-Datasets, אבל ניתן לראות בבירור ש- SSL_Connection מחזיק את הערך הגבוה ביותר שהוא 1 לעומת כל שאר הפ'יצרים ולכן יהיה ה- root

node בעץ ההחלטה. העלה הבא יהיה ה-google position מאחר שהוא הערך השני הגבוהה ביותר, כל צומת עם משקל קטן יותר ייתכן ולא יקבל התייחסות ע״י תהליך ה-pruning. וכמובן ה-submit אשר מכיל את הערך אפס לא ייכנס בכלל לעץ ההחלטה.

	BEFORE PRUNING				
Dataset	Accuracy	Precision	Recall	F-M	FP
Group A	99.71%	99.43%	100%	99.71%	2
Group B	98.29%	97.25%	97.25%	97.25%	6
Group C	99.29%	99.40%	99.60%	99.50%	3
Average	99.12%	98.69%	98.95%	98.82%	3.5
	AFTER PRUNING				
Dataset	Accuracy	Precision	Recall	F-M	FP
Group A	99.71%	99.43%	100%	99.71%	2
Group B	98.29%	97.25%	97.25%	97.25%	6
Group C	99.29%	99.40%	99.60%	99.50%	2
Average	99.12%	98.69%	98.95%	98.82%	3.3

ניתן לראות כי גם לפני וגם אחרי תהליך ה-pruning המדידות לא השתנו, ולכן הדיוק לא השתנה וזהו סימן טוב. השינוי העיקרי נעשה במספר ה-nodes ו-בכך מקטין את הסיבוכיות וזמן המימוש.

תהליך ה-Pruning:



בעקרון מהות התהליך בעצי החלטה הוא למחוק חלקים מהעץ אשר לא נותנים מספיק משקל בהחלטה של ה-classifier. התהליך מוריד את ה-complexity של ה-classifer הסופי, ואמור לשפר את הדיוק בעזרת הפחתה של overfitting (overfitting = התאמה גבוהה מידי של האלגוריתם לסט מצומצם של נתונים).

תהליך ה-pruning מבוצע ע"י מעבר על העלים כאשר כל צומת מוחלף עם המחלקה הפופולרית ביותר שלו. אם הדיוק לא משנה אזי השינוי נשמר.

השלב השני מכוון לבעיית בחירת ה-classifer הטוב ביותר מתוך הנבחרים אשר ישומשו במחקר. זה נעשה בעזרת אימון ובדיקה של classifers, אשר על האלגוריתמים הללו עברנו בסעיף הקודם (3.1.1).

בשלב הזה, תהליך "החזרת ה-dataset" יחזיר את שלושת הסטים אחד בתורו, ויעביר אותו לתהליך אימון ולידציה. החלק הכי חשוב במודל הוא ההפנייה ל-classifers אשר משומשים בכל סיבוב משלב היעילות ועד שלב האימון והולידציה.

לאחר מכן, נוצרו שלושה סטים של מידע (A,B,C) ועל מנת לקבל את התוצאות הטובות ביותר עברו על כל אחד מהאלגוריתמים (סעיף 3.1.1) וביצעו עליו את התהליך.

התוצאות שהתקבלו עבור דיוק ה-classifers היו להלן:

Individual Technique Precision				
SET	**C4.5**	**LR**	**KNN**	**SVM**
A	99.92%	99.92%	99.76%	99.92%
B	99.88%	99.88%	99.66%	99.88%
C	98.51%	98.51%	98.66%	98.51%

- SVM = Support vector machine
- KNN = K Nearest Neighbours
- LR = Logistic Regression

בשלב השלישי, יתבצעו ניסויים בשימוש בסטים של מידע על classifers אישיים בוצעו בשלב השני ועל בסיס הפלט שלהם. כלומר, בוצע שילוב של מכלול אלגוריתמים וניסיון למצוא את המכלול המוצלח ביותר שיזהה אתרי פישינג בצורה המוצלחת ביותר. בשלב הזה מאחר שהתבצע שימוש באלגוריתם הצבעה והיו 4 אלגוריתמים שעליהם התבצע הניסוי, הם חולקו לקבוצות של 3 על מנת שיצליחו להגיע לתוצאה הטובה ביותר.

מכלול האלגוריתמים חולק בצורה הבאה:

Ensemble	Alg1	Alg2	Alg3
Ensemble 1	KNN	C4.5	LR
Ensemble 2	KNN	C4.5	SVM
Ensemble 3	KNN	LR	SVM
Ensemble 4	C4.5	LR	SVM

כלומר, ל-4 קבוצות מכלול.

ותוצאותיו היו יותר מרשימות ממקודם שכן הצליחו להעלות את תוצאות כל הפרמטרים בכל סט של מידע. למשל בסט מידע הראשון התוצאות שופרו ממקודם שכן הפעם ה-Accuracy הצליח להגיע למשל ל-99.2% דיוק.

יתרונות וחסרונות דרך הפתרון במאמר

אחד היתרונות העיקריים בדרך הפתרון במאמר הוא כמובן השימוש ב-Machine Learning לזיהוי אתרי פישינג. ישנם כלים רבים אשר השתמשו בשיטה של Blacklist / Whitelist אשר הוכחו כאמינים פחות, בעיקר עקב אחד המחקרים הסטטיסטים שלפיהם נוצרים 1.4 מיליון אתרי פישינג חדשים מידי חודש.

החסרון העיקרי הוא שתוקפים / מזיקים למינהם יוכלו ללמוד את הפ'יצרים שעל פיהם התבסס המחקר, ולהרים אתר שיעמוד ב-"תנאי הלגיטימיות" אשר לא יזוהה כאתר פישינג. ולכן המרדף כאן הוא אינסופי, מהצד של המזיק ומהצד המגן.

מבוא לתהליך הוצאת הפ'יצרים

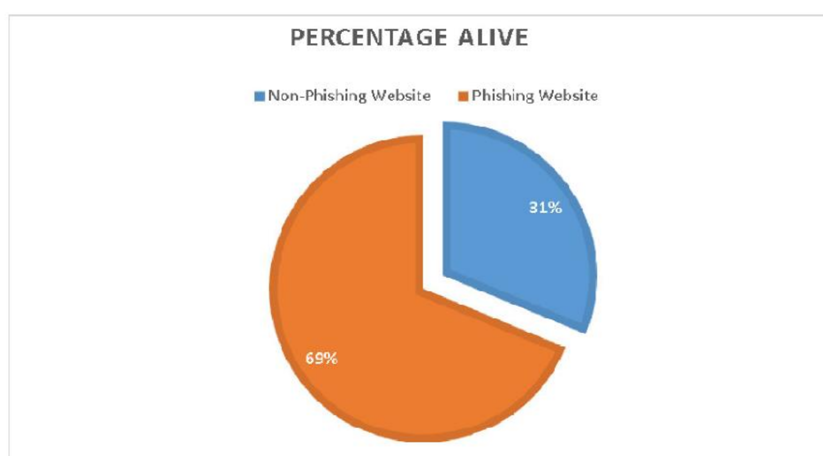
זהו התהליך שבו מתבצעת שליפה של פרמרטרים ודירוגם על מנת שאלגוריתם הלמידה יוכל לפיהם לחשב את האחוז הסטטיסטי שאתר מסוים הוא אתר פישינג. הרעיון הבסיסי הוא שהפלט של ההוצאה של הפרמטרים ישומש בתור קלט ל-Classifiers שונים.

כל אוספי המידע אשר משמשים להוצאת הפרמטרים במחקר, נלקחו מהמאגר של Phistank, והאתרים הלגיטימיים נלקחו מחיפושים בגוגל.

בהמשך הפרק ידובר על תהליך שליפת הפרמטרים כולו, אימות המידע, ונרמול המידע. בנוסף, מדובר על חלוקת המידע, כלומר, במימדים של איחוד נתונים ואחוזי האתרים אשר פישינג או אינם אתרי פישינג, על מנת להעלות את היעילות של תהליך אימון המסווגים ובכך להשיג תוצאות מדוייקות יותר.

התהליך נכתב ב-PHP, ולכן היה צורך להתאים את המידע שהורד מ-Phistank ל-SQL במקום CSV. ובוצעה הוספה ידנית (באמצעות סקריפט שנכתב ב-PHP) של פ׳יצרים נוספים למאגר מאחר שעל-פי טענתם מאחר כי Phistank חינמי, הוא לא סיפק את כל הפ׳יצרים שהם רצו להתייחס אליהם. הסטטיסטיקות של האתרים אשר נמצאו במאגר הינם:

	Phishing Websites	Non-Phishing Websites
Total Collected	7612	1638
Offline	3999	0
Tested Alive	3611	1638



ניתן לראות שבחלוקת המידע רוב האתרים במאגר הינם אתרי פישינג (69%) וישנם גם כ-31% שעליהם משתמשים על מנת שהאלגוריתם יוכל על פיהם מיהם האתרים החשודים ומי הם אלו שאינם. בסה״כ במאגר קיימים 7612 אתרי פישינג ו-1638 אתרים אשר אינם פישינג. כמחצית אתרי הפישינג לא היו באוויר בזמן כתיבת הספר.

תהליך ההוצאה

הרעיון מאחורי תהליך הוצאת הפ׳יצרים הוא למצוא את הסט האפקטיבי המינימלי של פ׳יצרים שניתן להשתמש בו על מנת לזהות אתרי פישינג. הסקריפט אשר שולף את הפרמטרים הנוספים רץ על המאגר מ-Phistank, בודק שהאתר באוויר, מתחבר אליו ומוציא ממנו את הפ׳יצרים הנוספים עבור כל אתר.

עבור אתרים אשר אינם אתרי פישינג, נכתב Crawler אשר משמש להוצאת סט של מידע מגוגל וגם בעזרת הוצאה ידנית אשר נעשית ע״י חיפוש במנוע חיפוש של גוגל והוצאת הקוד נעשית בעזרת קוד ב-.php

רשימת הפ׳יצרים שאליהם הולך להתייחס התהליך:

1. כתובת **URL** ארוכה - כתובות ארוכות יכולות להחביא חלקים חשודים בשורות הכתובות. ישנה דרך מדעית לנבא האם אורך של כתובת הוא אתר פישינג או אינו. ולכן זהו קריטריון אשר משמש בעזרת קריטריונים נוספים לזהות האם אתר מסויים הוא אתר פישינג. בפרוייקט הנוכחי חישבו ידנית אורכים של כתובות של אתרי פישינג וגילו שאורך שגדול מ-127 הוא כנראה בסבירות גבוהה יותר אתר פישינג.

2. נקודות - אתר מאובטח מכיל לכל היותר 5 נקודות. אם למשל יש יותר מ-5 נקודות בעמוד אז הוא יכול להיות מזוהה כאתר פישינג. למשל:

http://www.example.com.my/www.facebook.com/index.php

3. כתובת IP - חלק מהאתרי פישינג נשמרו ככתובות IP במקום בתור כתובת URL תקינה. זהו נקודה חשודה שכן כנראה לרוב האתרים הלגיטימיים לא יוצגו בצורה הזו. ובעיקר מאחר שרוב אתרי הפישינג לא נמצאים באוויר לתקופה ארוכה זהו מנגנון מאוד יעיל לזיהוי אתרים חשודים.

4. חיבור SSL - רוב האתרים שבהם מתבצע תשלום כלשהוא לרוב מוצפנים בעזרת תעבורת HTTPS. בנוסף, ניתן להשתמש בחתימה הזו על מנת לזהות את האתר אשר משתמש בחתימת SSL, אשר מכילה מידע ספיציפי על האתר.

5. שטרודל (@) - אתרי פישינג יכולים להכיל בסבירות גבוהה יותר את הסימן שטרודל (@) , בתוך שורת הכתובות שלהם, מאחר שהדפדפן מתי שקוראים אתרי אינטרנט מתעלים מכל מה שמשאל ל-@. (ארחיב ואומר כאן שזה גם תופס עם סימן הסולמית #).

6. - Hexdecimal פ׳יצר נוסף אשר ייחודי לאתרי פישינג הם כתובות מוצפים ע״י hex, מאחר כי דפדפנים תומכים בתצורה הזו של כתובות למשל:

bin/login.php-paypal/cgi/…http://%32%31%32%

7. הפניות - אפליקציות web מקבלות קלט מהמשתמש אשר מציין אתר חיצוני אשר משמש ללינק בהפניות. זה מקל על התקפות פישינג. למשל:

www.facebook.com;example.com

זה יבצא הפנייה בעזרת שימוש ב-facebook כאתר מפנה.

8. כפתור **ubmitS** - אתרי פישינג לרוב מכילים את כפתור ה- submit בקוד מקור שלהם, אשר לרוב מכיל כתובת אימייל של התוקף או למסד נתונים, מתי שהמשתמש חופש שהאתר הוא לגיטימי הוא מכניס פרטים אישיים ולוחץ על כפתור השליחה. לרוב העמוד או לא יכול להימצא או שישנה איזו שהיא הודעת שגיאה שמופיעה על המסך אשר מציינת שגיאת רשת כלשהיא. זה נפוץ בתהליכי שכפול אתרים.

תיאור הניסויים ומסקנותיהם

לאחר הוצאת הפרמטרים יתבצע תהליך של נרמול המידע, בעזרת נרמול מינימום-מקסימום המבצע טרנספורמציה לינארית על הערך שהתקבל. הנרמול מתבצע בשיטה הבאה (פרק 4 בספר): "נניח כי minA ו-maxA הם ערכי המינימום והמקסימום של ערך A. נרמול מקסימום-מינימום ממפה כל ערך של v of A לערך v' בטווח החדש בין המינימום והמקסימום החדש, אשר מחושב עם המשוואה הבאה":

$$v' = \left(\frac{(v - min_a)}{(max_a - min_a)}\right) * \left((new - max_a) - (new - min_a)\right) + new - min_a \qquad (4.2)$$

הפ'יצרים שהוצאות הם סט של ערכים אשר מתוארים בחוקיות הבאה: (כאשר: i=n, n={1,n+1,...9})

F(i) = { 1 if URL phishing | otherwise 0}

הצורך בנרמול המידע נובע מכך שהוא עוזר להגדיל את הדיוק של תוצאות. למשל עבור טווח ערכים של אורך URL מ-30 ל-300 יומר לטווח ערכים חדש של בין 0-1. אחרי עיבוד המידע, מחלקים את המידע ל-3 סטים של בדיקות על מנת לבדוק באיזו שיטה יתקבל הדיוק הגבוה ביותר.

הסט הראשון הוא חילוק של 70% אתרי פישינג ו-30% אתרי לא פישינג הסט השני הוא חלוקה שווה של 50% מכל אחד והסט השלישי הוא חילוק של 30% אתרי פישינג ו-70% לא אתרי פישינג. בנוסף, יש שימוש ב-10% של מידע מעורב על מנת להעריך את היעילות של כל אחד מהסטים.

מבוא למימוש ולתוצאות המחקר

היעידים המרכזיים בפרק זה הם האימון והבדיקות של האלגוריתמים ה-"מסווגים" ,K Nearest Neighbour Logistic Regression, Support Vector Machine & Decision Tree.

ייעשה שימוש באותו סט של מידע עבור כל אחד מהאלגוריתמים על מנת לבדוק מי הוא האלגוריתם בעל והדיוק הגבוה ביותר. אחד מהסיבות העיקריות לדיוק נמוך הוא בחירה של פ'יצרים בעלי משקל נמוך לאלגוריתמים.

תהליך המחקר מחולק ל-3 חלקים עיקריים, אימון ובדיקה, תכנון והרכבה ובסוף פתרון הניתן להשוואה. אימון המודל והבדיקה שלו מתבצע על-ידי אימון יחיד של כל אחד מהאלגוריתמים שנבחרו. תהליך התכנון וההרכבה מתבצע על-ידי בחירת מספר אלגוריתמים. כלומר, ייבחרו 3 אלגוריתמים כל פעם בהרכב טיפה שונה מתוך 4 האלגוריתמים, וע"י אלגוריתם הצבעה נוכל להגיע לתוצאות טובות יותר בעזרת שימוש בעזרת קבוצה של 3 מסווגים במקום.

תוצאות הניסוי ומסקנותיו

במהלך הניסוי נבחנו כל ארבעת האלגוריתמים על מנת לבדוק את יעילותם. אלגוריתם K-Nearest Neighbours נבדק עם מספר שכנים משתנה מ-1 עד 7, על מנת לבדוק את יעילות האלגוריתם והאם מספר השכנים משפר את יעילות האלגוריתם. ואכן כך ייצא נראה כי אחוז הדיוק ירד כאשר K-NN1 = 99.37% וב-K-NN7 = 98.97%. כלומר בעזרת שכן בודד נקבל את התוצאות הטובות ביותר.

במהלך המימוש, נעשה שימוש במספר פרמטרים שונה על מנת לבצע אימון ובדיקה של האלגוריתמים על מנת להצדיק את הפרמטרים אשר השתמשו בהם. תהליך עיקרי היה להחליף מספר פעמים במהלך התהליך הראשי את מספר הולידציות.

במהלך הניסוי התבצע שימוש ב-3 סטים של מידע שונים, והיה נראה כי הסט הראשון A, עם האלגוריתם K-NN1 קיבל את אחוז הדיוק הגבוהה ביותר 99.37% במהלך המחקר הם החשיבו גם את ה-Accuracy, Precision וגם ה-f-measure על מנת למדוד את יעילות האלגוריתמים וקיבלו כי האלגוריתם K-NN היה זה שתפקד הטוב ביותר מכולם. לא רק זאת, גם אחוז ה-"התראת שווא" היה הנמוך ביותר משמעותית באלגוריתם זה.

לאחר מציאת האלגוריתם הבודד היעיל ביותר, הגיע השלב של הרכבת האלגוריתמים יחדיו על מנת לבדוק האם ניתן להגיע לתוצאות טובות יותר. מאחר שנעשה שימוש באלגוריתם הצבעה, היה צורך בשימוש במספר אי-זוגי של אלגוריתמים בכל הרכבה של קבוצת אלגוריתמים אשר משתתפים בהצבעה. ולכן הורכבו 4 קבוצות שונות אשר מכילות כל אחד הרכב שונה של 3 אלגוריתמים מתוך 4 הנבחרים. נראה כי על בסיס התוצאות שההרכבה תפקדה טוב ביותר כאשר הסט של המידע חולק שווה בשווה בין אתר פישינג לאתרים אשר אינם אתרי פישינג (50-50). ומאחר כי כל ההרכבות הראו תוצאות שוות כאשר נעשה שימוש בסט B, נראה כי ניתן להסיק שכל אחד מההרכבות יכול להיות בשימוש. הגרף הבא לקוח מהספר והוא משווה בין האלגוריתם הבודד הטוב ביותר K-NN לבין האלגוריתם ההרכבה הטוב ביותר:

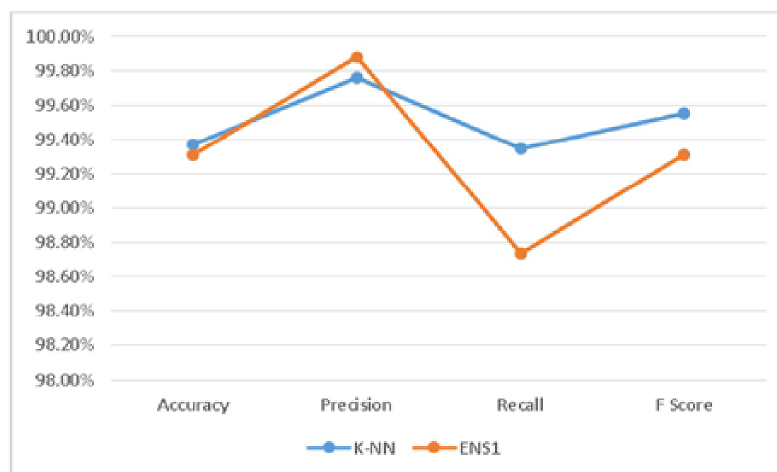
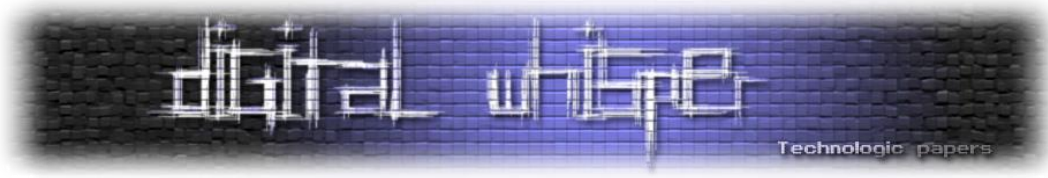


Figure 5.18: Plot of best individual against best ensemble algorithm

על פי הגרף, למרות שניתן לראות כי האלגוריתם הבודד מקבל Accuracy טיפה יותר גבוהה מאלגוריתם ההרכבה הטוב ביותר, אך ה-Precision גבוהה יותר באלגוריתם ההרכבה. ולכן ניתן להסיק כי אלגוריתם ההרכבה יכול לשמש לשיפור הביצועים של האלגוריתמים הבסיסיים.

נראה כי הפי'צרים שנבחרו במאמר היו חזקים מידי שכן שינוי באחוזי הלמידה/וולידציה לא הראה שינוי גדול ב-accuracy של המודל. וזה נראה לא סביר שכן בוודאי ל-70% לימוד ו-30% ולדיציה אמור להיות דיוק גבוהה יותר משמעותית מ-30% לימוד ו-70% וולידציה.



פרק זה מתבסס על המאמר New Rule-Based Phishing Detection, המחקר והניסויים במאמר נעשו על מנת לזהות אתרי Banking מתחזים (Phishing), אשר מנסים לגנוב סיסמאות מלקוחות הבנקים. אלגוריתם הלמידה אשר נעשה בו שימוש במאמר הוא Supported Vector Machine. בנוסף של דבר הם הטמיעו את החוקיות ב-browser extension אשר מזהה ב-Live אתרי פישינג.

תיאור סט הנתונים

השיטה שבה נעשה שימוש במאמר היא שימוש ב-2 סטים של פ'יצרים. הסט הראשון, הוא סט "מוצע" אשר הוצע ע"י החוקרים במאמר, והסט השני הוא סט של פ'יצרים אשר נלקח ממחקרים קודמים וצמצם לסט של 5 פיצרים מתוכם.

הסט הראשון ה-"מוצע" של הפיצרים האלה מכילים 4 פיצרים שמעריכים את הזהות של העמוד, ו-4 פיצרים שמזהים את פרוטוקול הגישה לאלמנטים בעמוד. שני הפ'יצרים האלה נשלפים בעזרת ניתוח ואבחון ה-DOM (Document Object Model). בעזרת שימוש ב-approximate string matching algorithm (fuzzy search - חיפוש שמחפש pattern, ולא בדויק את הערך). בנוסף, נעשית התחשבות האם העמוד מאובטח או לא.

הסט השני אשר נקלח ממחקרים קודמים, צומצם ל-5 פיצרים בלבד אשר החוקרים העריכו כאשר אלו אשר יביאו את הערך הגדול ביותר. והם: כתובת אייפי, SSL, מספר הנקודות ב-URL, אורך הכתובת, ומילים מרשימה שחורה.

סט המידע נלקח מהאתר Yahoo direct service, על מנת לספק אתרים לגיטימיים והמידע של אתרי פישינג נלקח מהאתר phistank. אוסף המידע הכיל אתרי פישינג אשר מתחזים לאתרי בנקים או אתרי בנקים לגיטימיים.

תיאור הניסויים ומסקנותיהם

אחרי שלב העבודה על הפ'יצרים נותר להגיע לשלב של "סיווג" העמוד, כלומר הזיהוי שלו האם הוא אתר פישינג או אתר לגיטימי. האלגוריתם שבו נעשה שימוש הוא SVM אשר הוא אלגוריתם מסווג אשר נמצא בשימוש רחב בשנים האחרונות.

הלמידה והבדיקה התבצע על שלושה סטים של פ'יצרים, הראשון הכיל את תשעת הפיצרים הראשונים (FS1). השני הכיל רק את ה-"מוצעים" (FS2), והשלישי הכיל את כל הפיצרים מהספרות וגם המוצעים אשר התרכזו בסה"כ 17 פיצרים (FS3).

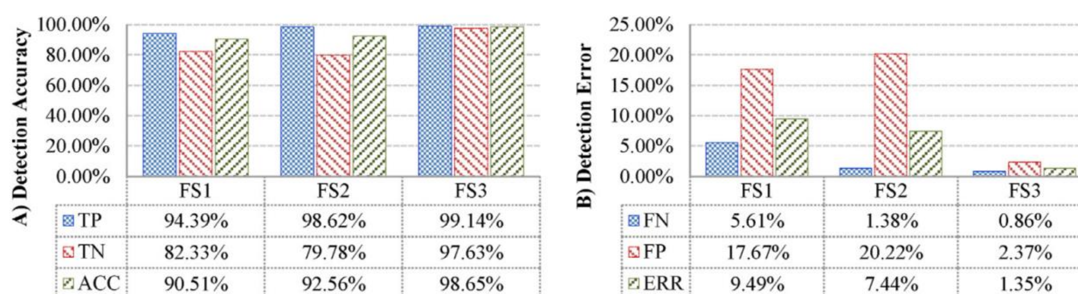


Fig. 4. Result of classification on each feature set, (A) Detection accuracy, (B) Detection error.

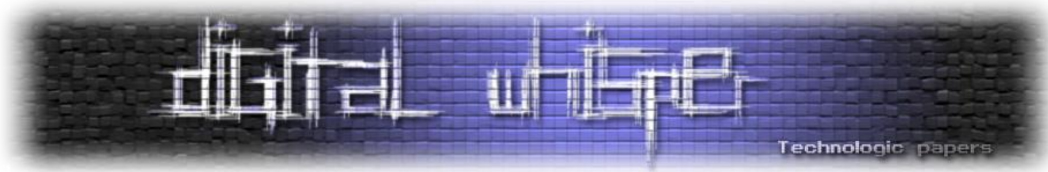
TP - True Positive, TN - True Negative, ACC – Accuracy

Table 4
The result of classification with SVM.

Classification output	Value
Correctly classified instances	98.65%
Incorrectly classified instances	1.35%
Kappa statistic	0.969
Sensitivity	0.9914
F-Score	0.9901
Root mean squared error (RMSE)	0.102

FN - False Negative, FP - False Positive, ERR - Error

השימוש בסט מידע השלישי (FS3), גרם לדיוק של זיהוי אתרי הפישינג כ-TP לעלות ל-99.14%. וגם הערך של FN אשר שיחק תפקיד חשוב ירד גם כן ל-0.86%.



יתרונות וחסרונות דרך הפתרון במאמר

נראה כי במאמר הצליחו החוקרים להגיע לתוצאות יפות של 98.65% זיהוי אתרי בנקים מתחזים כאתרי פישינג. ונראה כי עצם השימוש ב-Browser Extension על מנת להטמיע. את הלוגיקה וזיהוי אתרים חשודים בעת גלישה באינטרנט "לייב" נמצא מאוד שימושי.

חלק מן היתרונות במאמר הם כמובן הפ'יצרים שהוצעו ע"י החוקרים, בנוסף לפ'יצרים שהם החליטו להשתמש בהם ממחקרים קודמים. נראה כי פ'יצרים אלו שיחקו תפקיד חשוב בהגעה לאחוז ההצלחה הגבוהה בזיהוי אתרי פישינג.

בנוסף השימוש במנגוני למידת מכונה כמובן מראה יתרון גדול על שיטות אחרות כמו למשל אתרי blacklist שכן יכול לזהות zero-day attacks עוד לפני שאתרים אלו נכנסו לרשימה השחורה.

החסרונות בדרך הפתרון במאמר הם כמובן, מאחר שיש התבססות גדולה על ה-DOM ועל המבנה שלו. נראה כי אם המשתמש יכניס קבצי SWF, Image ככאלו שיציגו את האתר במקום ב-HTML, אזי כנראה שהאלגוריתם עם הפי'צרים הללו יתקשה לזהות האם אתר מסויים הוא פישינג.

בנוסף המאמר בא מנקודת הנחה שתוקף לא ישקיע זמן רב בעיצוב עמוד הנחיתה אלא בדר"כ רק יעתיק את ה-HTML ויתבסס עליו. ולכן, במקרה שהתוקף יעבוד על העמוד זה יוכל להקשות על האלגוריתם גם במקרה הזה.

מימוש להמחשה

כל הפרוייקט מתועד ב-GitHub:

https://github.com/danrevah/detect-phishing-websites

תיאור הסביבה וסט הנתונים

בחלק הזה, נכתוב סקריפט ב-Python אשר ביצע את טעינת מידע ל-turicreate, ובעזרת אלגוריתמי למידת מכונה שונים נבצע ניסוי שבו ננסה לראות איזה אלגוריתם מחזיר את התוצאה המדוייקת ביותר מביניהם.

הבחירה ב-Python בתור שפה המשמשת לחישובים ולכתיבת קוד מהיר הייתה נראית נכונה יותר, נראה מאחר כי קהילת ה-Machine Learning, אימצה את Python בתור שפת פיתוח ומכאן שניתן למצוא בה ספריות Open-Source די בקלות. בנוסף, הכוח של Python מגיע בביצועים כאשר מבצעים פעולות חישוביות מורכבות, כמו שנדרש כאשר מריצים אלגוריתמי למידת מכונה.

turicreate, הוא פרוייקט אשר נקנה לאחרונה ע״י Apple, ונכתב ב-python, נראה כי זהו פרוייקט די מפורסם בקהילת ה-Open Source עפ״י הדירוג שלו ב-Github, כמות ה-Contributers והמאמרים שניתן למצוא עליו באינטרנט. בעיקרון הוא עוזר לפשט את תהליך הפיתוח של התאמת מודלים ללמידת מכונה, ונראה מתאים יותר לחבר'ה שהם לא מומחים בתחום ה-Machine Learning ולכן נראה לי כבחירה נכונה להשתמש בו לפיתוח.

סט המידע לניסוי נלקח מהאתר "Machine Learning Repository, ומהמאגר הבא:

https://archive.ics.uci.edu/ml/datasets/Phishing+Websites

האלגוריתמים שנבחרו

בחרתי מספר אלגוריתמים להשוואה על מנת לבדוק איזה אלגוריתם יביא לתוצאה המדוייקת ביותר. האלגוריתמים שנבחרו לצורך הניסוי הם:

1. **Boosted Decision Tree -** מודל זה מבצע חישובים וחיזוי על-פי איחוד של החלטות מרצף של מודלים בסיסיים.
2. **Decision Tree -** מודל זה מבוסס על מבנה של תרשים זרימה, כאשר כל צומת (שאינה עלה), מבצעת ניסוי על ערך כלשהוא, וכל אחד מהענפים מייצג את התוצאה של הניסוי, וכל עלה (או צומת סופי), מחזיק את תווית המחלקה. הצומת שנמצא בראש העץ הוא צומת הבסיס.
3. **K Nearest Neighbour -** מודל זה חוזה את המחלקה הכי נפוצה מתוך נקודות השכנים. .
4. **Linear Regression -** רגרסיה לינארית מנסה להשתמש בנוסחה על מנת לייצר ערך אמיתי. הפונקציה הזו משתמשת בערך דיסקריטי על מנת לחזות ערכים.

תהליך הפיתוח

השלב הראשון היה להוריד את קובץ ה-arff מהאתר:

https://archive.ics.uci.edu/ml/datasets/Phishing+Websites

ולהמיר אותו לקובץ CSV, על מנת שיהיה קריא ל-turicreate. הקובץ הכיל את הפ'יצרים הבאים:

- having_IP_Address { -1,1 }
- URL_Length { 1,0,-1 }
- Shortining_Service { 1,-1 }
- having_At_Symbol { 1,-1 }
- double_slash_redirecting { -1,1 }
- Prefix_Suffix { -1,1 }
- having_Sub_Domain { -1,0,1 }
- SSLfinal_State { -1,1,0 }
- Domain_registeration_length { -1,1 }
- Favicon { 1,-1 }
- port { 1,-1 }
- HTTPS_token { -1,1 }
- Request_URL { 1,-1 }
- URL_of_Anchor { -1,0,1 }
- Links_in_tags { 1,-1,0 }
- SFH { -1,1,0 }
- Submitting_to_email { -1,1 }
- Abnormal_URL { -1,1 }
- Redirect { 0,1 }
- on_mouseover { 1,-1 }
- RightClick { 1,-1 }
- popUpWidnow { 1,-1 }
- Iframe { 1,-1 }
- age_of_domain { -1,1 }
- DNSRecord { -1,1 }
- web_traffic { -1,0,1 }
- Page_Rank { -1,1 }
- Google_Index { 1,-1 }
- Links_pointing_to_page { 1,0,-1 }
- Statistical_report { -1,1 }
- Result { -1,1 }

בסוגריים המסולסלים מיוצג טווח הערכים של כל עמודה. על-פי הפ'יצרים האלה turicreate יבצע הסקה בעזרת כל אלגוריתם האם אתר מסויים הוא אתר חשוד או לא. העמודה האחרונה, Result מייצגת את התוצאה (1- אתר פישינג, אחרת אתר לגיטימי). שורות לדוגמא מקובץ המאגר:

```
@data
-1,1,1,1,-1,-1,-1,-1,-1,1,1,-1,1,-1,1,-1,-1,-1,0,1,1,1,1,-1,-1,-1,-1,1,1,-1,-1
1,1,1,1,1,-1,0,1,-1,1,1,-1,1,0,-1,-1,1,1,0,1,1,1,1,-1,-1,0,-1,1,1,1,-1
1,0,1,1,1,-1,-1,-1,-1,1,1,-1,1,0,-1,-1,-1,-1,0,1,1,1,1,1,-1,1,-1,1,0,-1,-1
1,0,1,1,1,-1,-1,-1,1,1,1,-1,-1,0,0,-1,1,1,0,1,1,1,1,-1,-1,1,-1,1,-1,1,-1
```

מבט קצר על הקוד

```python
import sys
import turicreate as tc

def execute(algo):
    data =  tc.SFrame('dataset.csv')
    train_data, test_data = data.random_split(0.8)

    if algo == 1:
        model = tc.boosted_trees_classifier.create(train_data, target='Result', max_iterations=2, max_depth = 3)
    elif algo == 2:
        model = tc.decision_tree_classifier.create(train_data, target='Result')
    elif algo == 3:
        model = tc.nearest_neighbor_classifier.create(train_data, target='Result')
    else:
        model = tc.logistic_classifier.create(train_data, target='Result')

    predictions = model.classify(test_data)
    return model.evaluate(test_data)

try:
    print("[1] Boosted decision tree\n"
          "[2] Decision tree\n"
          "[3] Nearest neighbour\n"
          "[4] Logistic regression")

    num = int(input("Select ML Algorithm: "))

    if not num or num < 1 or num > 4:
        raise ValueError()

    results = execute(num)
    print("\n\n\n>>> Accuracy\t: %s" % results['accuracy'])
    print(">>> Precision\t: %s" % results['precision'])
    print(">>> Recall\t: %s" % results['recall'])
    print(">>> F1 Score\t: %s" % results['f1_score'])

except ValueError:
    print("\nError: Please select a number between 1 to 3.")
    sys.exit(1)
```

כפי שניתן לראות, הקוד שנכתב הוא די מצומצם וזה תודות ל-turicreate אשר מקל על כל תהליך הפיתוח.

הסקריפט נותן למשתמש לבחור אלגוריתם לבדיקה, לאחר מכן קורא למתודה 'exeute', ומריץ את האלגוריתם שנבחר על 80% מהמידע, ובודק מול 20% את הדיוק של המודל שנוצר.

תוצאות הריצה

הסקריפט שנכתב נותן למשתמש לבחור מבין ארבעת אלגוריתמי למידת מכונה שציינו בסעיפים הקודמים. מכאן בעזרת turicreate, הוא לוקח את קובץ סט הנתונים dataset.csv מחלק אותו ליחס של 80% סט נתונים לאימון ו-20% סט נתונים לולידציה. לאחר מכן, הוא טוען ומבצע ריצה על 30 העמודות הראשונות, העמודה הימנית ביותר זאת העמודה אשר מציינת אם האתר הוא אתר Phishing או לא.

ניתן לראות ש-turicreate מספק מידע רב לגבי תהליך הריצה, ולאחר מכן בסוף אנו מדפיסים את ה-Accuracy של האלגוריתם הנבחר.

בתמונה הבאה נראה תוצאת ריצה, כאשר אנו בוחרים באלגוריתם Boosted Decision Tree, ומתבצעת חלוקה ל-20% ו-80% בעזרת ערבוב ראנדומלי עם ה-80% מתבצע לימוד האלגוריתם ובעזרת ה-20% מתבצעת הולידציה.

הפלט שמתקבל על המסך הוא הפלט הסטנדרטי של turicreate והוא מציג את מבנה העמודות. ניתן לראות שהוא מזהה את סוג כל העמודה עפ״י 100 השורות הראשונות, במקרה שלנו כולם int.

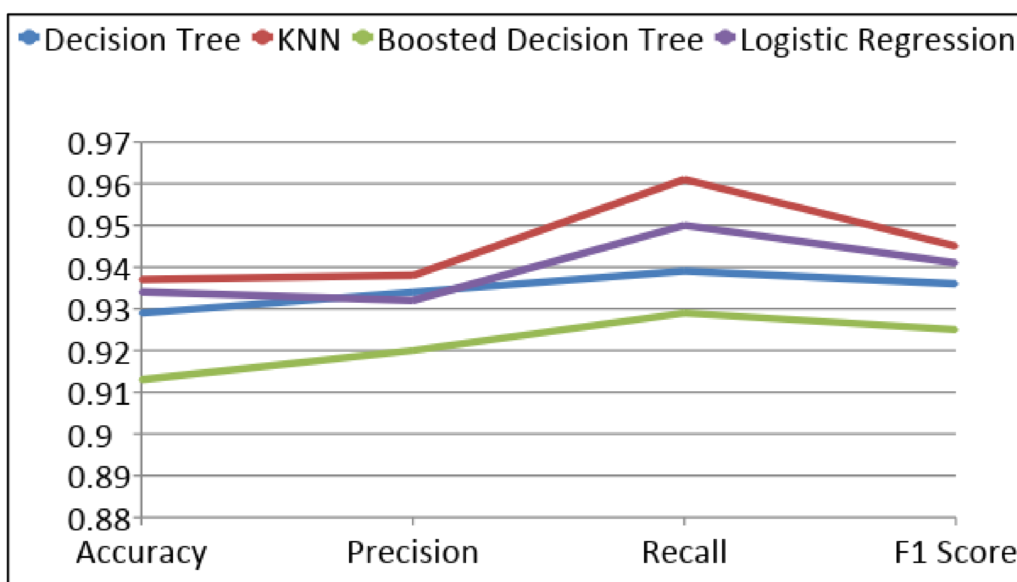
לאחר מכן, הוא מדפיס פרטים נוספים הנוגעים לאלגוריתם boosted tree classifier בין היתר מספר הדוגמאות, מספר המחלקות, מספר עמודות הפיצרים. לאחר מכן מתבצעת הדפסה של כל הפרמטרים שאנו משתמשים בהם על מנת להעריך את איכות האלגוריתם: accuracy, precision, recall, f1_score

```
1. danrevah@Dans-MacBook-Pro: ~/dev/openu/machine-learning (zsh)
[1] Boosted decision tree
[2] Decision tree
[3] Nearest neighbour
[4] Logistic regression
Select ML Algorithm: 1
Finished parsing file /Users/danrevah/dev/openu/machine-learning/dataset.csv
Parsing completed. Parsed 100 lines in 0.044112 secs.
------------------------------------------------------
Inferred types from first 100 line(s) of file as
column_type_hints=[int,int,int,int,int,int,int,int,int,int,int,int,int,int,int,int,int,int,int,int,int,int,int,int,int,int,int,int,int,int,in
t]
If parsing fails due to incorrect types, you can correct
the inferred type list above and pass it to read_csv in
the column_type_hints argument
------------------------------------------------------
Finished parsing file /Users/danrevah/dev/openu/machine-learning/dataset.csv
Parsing completed. Parsed 11055 lines in 0.030973 secs.
PROGRESS: Creating a validation set from 5 percent of training data. This may take a while.
          You can set ``validation_set=None`` to disable validation tracking.

Boosted trees classifier:
--------------------------------------------------------
Number of examples          : 8392
Number of classes           : 2
Number of feature columns   : 30
Number of unpacked features : 30
+-----------+--------------+-------------------+---------------------+-------------------+---------------------+
| Iteration | Elapsed Time | Training-accuracy | Validation-accuracy | Training-log_loss | Validation-log_loss |
+-----------+--------------+-------------------+---------------------+-------------------+---------------------+
| 1         | 0.012081     | 0.909676          | 0.906000            | 0.505258          | 0.507134            |
| 2         | 0.019812     | 0.918494          | 0.910000            | 0.401508          | 0.407070            |
+-----------+--------------+-------------------+---------------------+-------------------+---------------------+


>>> Accuracy    : 0.9135460009246417
>>> Precision   : 0.9184873949579831
>>> Recall      : 0.9239222316145393
>>> F1 Score    : 0.921196797302992
➜ machine-learning git:(master)
```

במספר ריצות התקבלו התוצאות הבאות:

	Neighbour Nearest K	Tree Decision	tree decision Boosted	Regression Logistic
Accuracy	0.937	0.929	0.913	0.934
Precision	0.938	0.934	0.920	0.932
Recall	0.961	0.939	0.929	0.950
Score F1	0.945	0.936	0.925	0.941

נציג את התוצאות שהתקבלו בגרף:



ניתן לראות שגם אצלנו המנצח בכל הפרמטרים ייצא K-NN בעל שכן בודד (כמו שהתקבל במחקר הראשון).

האלגוריתמים Decision Tree ו-Boosted Decision Tree תפקדו פחות טוב מאחר כי הערכים אינם בינאריים. כלומר, ייתכן 3 ערכים שונים לעמודה בודדת. נראה כי מאחר שישנה חלוקה של 3 סטים של מידע בכל עמודה התקבלו תוצאות פחות מדוייקות.

מסקנות וסיכום

נראה כי תהליך הפיתוח ב-Machine Learning, הופך לקל יותר עם השנים ואין צורך להיות מומחה ל-Machine Learning, על מנת להתחיל להיעזר בספריות כמו turicreate על מנת להסיק מידע על נתונים ולבצע עליהם למידת מכונה.

המסקנה העיקרית בתהליך המחקר וכתיבת הסקריפט היא שהבעיה העיקרית היא בעצם זיהוי הפ'יצרים שצריך להוציא ובחירת האלגוריתם הנכון לבעיה.

במאמר הראשון שקראתי, הם השתמשו בשיטה מתוחכמת יותר מאשר לבדוק כל אלגוריתם בנפרד. במקום לבצע הרצה מקבילה של כל אחד מן האלגוריתם ובדיקה איזה מהם מביא את התוצאה הטובה ביותר, הם הריצה את כל ארבעת האלגוריתמים וקיבלו את הדיוק של כל אחד בנפרד.

לאחר מכן, הם הרכיבו קבוצה של 3 אלגוריתמים בו-זמנית של כל אחת מ-4 האלגוריתמים שהם בחרו, והשתמשו באלגוריתם "הצבעה" שיצביע האם תוצאה היא נכונה. ובכך הגיעו לתוצאות מדוייקות יותר מאשר שימוש באלגוריתם בודד.

נראה כי, זוהי שיטה מעולה לשפר את דיוק הניסוי, ולדעתי צריך לשאוף אליה כאשר מבצעים תהליכי פיתוח בתחום.

ביבליוגרפיה

1. Akanbi, O., Abunadi, A., & Zainal, A. (2014). Phishing website classification: A machine learning approach. Journal of Information Assurance and Security, 9(5), 222-234.

2. I. S. Amiri, O. A. Akanbi, and E. Fazeldehkordi, A Machine-learning Approach to Phishing Detection and Defense. Syngress, 2014.

3. M. Moghimi and A. Y. Varjani, "New rule-based phishing detection method, "Expert Systems with Applications, vol. 53, pp. 231–242, 2016.

4. Online tech dictionary for students, educators and IT professionals: https://www.webopedia.com/

5. How SVM (Support Vector Machine) algorithm works: https://www.youtube.com/watch?v=1NxnPkZM9bc

6. Decision Tree 1 - how it works: https://www.youtube.com/watch?v=eKD5gxPPeY0

7. A Quick Introduction to K-Nearest Neighbors Algorithm: https://medium.com/@adi.bronshtein/a-quick-introduction-to-k-nearest-neighbors-algorithm-62214cea29c7

8. Accuracy, Precision, Recall or F1: https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb

לקבל מושג ירוק על קוברנטיס (Kubernetes)

מאת ליאור בר-און

הקדמה

המאמר הזה נכתב לאנשי-תוכנה מנוסים, המעוניינים להבין את - Kubernetes בהשקעת זמן קצרה. הבאזז מסביב ל-Docker ו-Docker Orchestration הוא כרגע רב מאוד. דברי שבח רבים מסופרים על הטכנולוגיות הללו, מבלתי להתייחס לפרטים ועם מעט מאוד ראיה עניינית וביקורתית. כאנשי-תוכנה ותיקים אתם בוודאי מבינים שהעולם הטכנולוגיה מלא Trade-offs, וכדאי לגשת לטכנולוגיות חדשות עם מעט פחות התלהבות עיוורת - וקצת יותר הבנה.

המאמר פורסם במקור כפוסט בבלוג "Software Archiblog - בלוג ארכיטקטורת תוכנה" מאת ליאור בר-און.

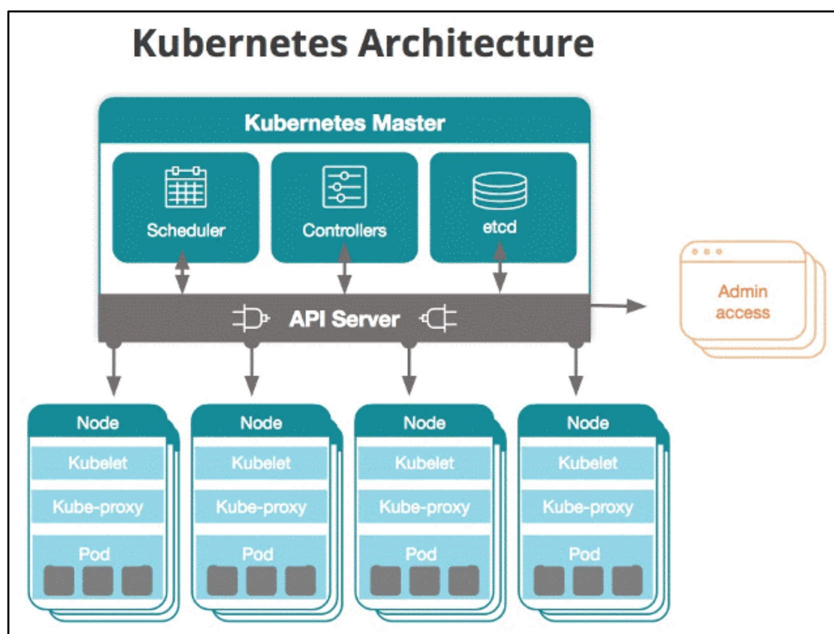
אני הולך לספק את הידע במאמר לא בפורמט ה- "From A to Z" כפי שדיי מקובל - אבל ע"פ סדר שנראה לי יותר נכון והגיוני. אני רוצה לדלג על דיון ארוך על ההתקנה, ועוד דיון ארוך על הארכיטקטורה - עוד לפני שאנחנו מבינים מהי קוברנטיס. אני הולך לדלג על כמה פרטים - היכן שנראה לי שהדבר יתרום יותר להבנה הכללית.

אז מהי בעצם קוברנטיס? פורמלית קוברנטיס היא Container Orchestration Framework (הנה פוסט שלי על הנושא) שהפכה בשנתיים האחרונות לסטנדרט דה-פאקטו (הנה פוסט אחר שלי שמנתח את העניין).

בכדי לפשט את הדברים, ניתן פשוט לומר שקוברנטיס היא סוג של סביבת ענן:

- אנו אומרים לה מה להריץ - והיא מריצה. אנו "מזינים" אותה ב-Containers (מסוג Docker או rkt) והגדרות - והיא "דואגת לשאר".
- אנו מקצים לקוברנטיס כמה שרתים (להלן "worker nodes" או פשוט "nodes") שהם השרתים שעליהם ירוצו הקונטיינרים שלנו. בנוסף יש להקצות עוד כמה שרתים (בד"כ - קטנים יותר) בכדי להריץ את ה-master nodes - ה"מוח" מאחורי ה-cluster.

- קוברנטיס תדאג ל-containers שלנו:
  - היא תדאג להרים כמה containers בכדי לאפשר high availability - ולהציב אותם על worker nodes שונים.
  - אם יש עומס עבודה, היא תדאג להריץ עוד עותקים של ה-containers, וכשהעומס יחלוף - לצמצם את המספר. מה שנקרא auto-scaling.
  - אם container קורס, קוברנטיס תדאג להחליף אותו ב-container תקין, מה שנקרא גם auto-healing.
  - קוברנטיס מספקת כלים נוחים לעדכון ה-containers לגרסה חדשה יותר, בצורה שתצמצם למינימום את הפגיעה בעבודה השוטפת - מה שנקרא deployment.
    - כפי שראינו בפוסט על Docker - פעולת restart של Container תהיה מהירה משמעותית מ-VM, שזה גם אומר לרוב deployments מהירים יותר.
  - לשימוש בקוברנטיס יש יתרון בצמצום משמעותי של ה-Lock-In ל-Cloud Vendor[1], והיכולת להריץ את אותה תצורת "הענן" גם On-Premises.
    - הסתמכות על קוד פתוח, ולא קוד של ספק ספציפי - הוא גם יתרון, לאורך זמן, וכאשר הספק עשוי להיקלע לקשיים או לשנות מדיניות כלפי הלקוחות.
  - קוברנטיס גם מספקת לנו מידה רבה של Infrastructure as Code, היכולת להגדיר תצורה רצויה לתשתיות רשת, אבטחה ועוד - מה שמייתר כלי ניהול תצורה (Provisioning) כגון Chef, Puppet או Ansible.



[1] אל דאגה! לספקי הענן יש אינטרס עליון לגרום לנו ל-Lock-In גם על סביבת קוברנטיס. לאחר שהניסיונות להציע חלופות "מקומיות" לקוברנטיס כשלו - רק טבעי שהם יתמקדו בלהציע יכולות שיפשטו את השימוש בקוברנטיס, אך גם יוסיפו סוגים חדשים של Lock-In. בכל מקרה, ברוב הפעמים אנו כבר תלויים בתשתיות כמו S3, Athena, RDS ועוד - ולכן כבר יש Lock-In מסוים גם בלי קשר לקוברנטיס. "Cloud Agnostic Architecture" הוא בסה"כ מיתוס, השאלה היא רק מידת התלות.

יעילות

עם כל היעילות המוגברת שהתרגלנו אליה מריצה בענן בעזרת שירותים כמו AWS EC2 או Azure Virtual Machines (להלן "ענן של מכונות וירטואליות") - קוברנטיס מאפשרת רמה חדשה וגבוהה יותר של יעילות בניצול משאבי-חומרה.

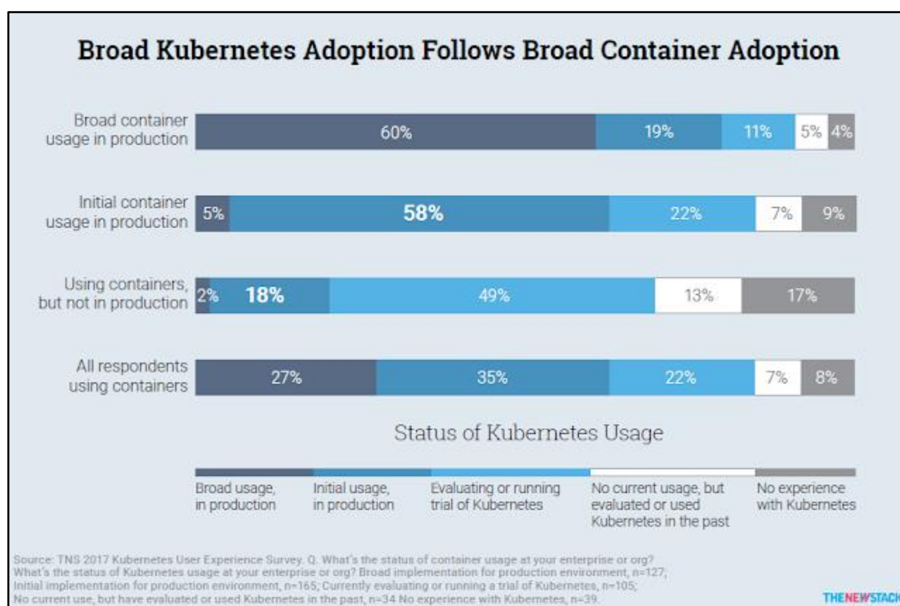
בתצורה קלאסית של מיקרו-שירותים, הפופולרית כיום - ייתכן ומדובר בניצולת חומרה טובה בכמה מונים. הכל תלוי בתצורה, אבל דיי טיפוסי להריץ את אותו ה-workload של מיקרו-שירותים בעזרת קוברנטיס על גבי 20%-50% בלבד מהחומרה שהייתה נדרשת על גבי "ענן ציבורי של מכונות וירטואליות".

איך זה קורה? למכונה וירטואלית יש overhead גבוה של זיכרון (הרצת מערכת ההפעלה + hypervisor) על כל VM שאנו מריצים. זה לא כ"כ משמעותי כשמריצים שרת גדול (כיום נקרא בבוז: Monolith) - אך זה מאוד משמעותי כאשר מריצים שרתים קטנים (להלן: מיקרו-שירותים).

מעבר לתקורה הישירה שעתה ציינו, יש תקורה עקיפה וגדולה יותר: כאשר אני מריץ על שרת 4 מיקרו-שירותים בעזרת VMs ומקצה לכל אחד מהמיקרו-שירותים 25% מהזיכרון וה CPU ההגבלה היא קשיחה. אם בזמן נתון שלושה מיקרו-שירותים משתמשים ב-10% ממשאבי המכונה כ"א, אבל המיקרו-שירות הרביעי זקוק ל-50% ממשאבי המכונה - הוא לא יכול לקבל אותם. ההקצאה של 25% היא קשיחה ואינה ניתנת להתגמשות, אפילו זמנית[2].

בסביבת קוברנטיס ההגבלה היא לא קשיחה: ניתן לקבוע גבולות מינימום / מקסימום ולאפשר מצב בו 3 מיקרו-שירותים משתמשים ב-10% CPU ו/או זיכרון כ"א, והרביעי משתמש ב-50%. אפשר שגם 10 דקות אח"כ המיקרו-שירות הרביעי יהיה idle - ומיקרו-שירות אחר ישתמש ב-50% מהמשאבים.

[2] שווה לציין שזה המצב בענן ציבורי. כששכן שלנו למכונה ב-AWS רוצה יותר CPU - למה שנסכים לתת לו? אנחנו משלמים על ה-"slice" שלנו במכונה - שהוגדר בתנאי השירות. בפתרונות של ענן פרטי (כמו VMWare) ישנן יכולות "ללמוד" על brusts של שימוש בקרב VM ולהתאים את המשאבים בצורה יעילה יותר. כלומר: המערכת רואה ש-VM מספר 4 דורש יותר CPU אז היא משנה, בצורה מנוהלת, את ההגדרות הקשיחות כך שלזמן מסוים - הוא יקבל יותר CPU מה VMs האחרים הרצים על אותה המכונה. טכנולוגית ה-VM עדיין מקצה משאבים בצורה קשיחה - אך תכנון דינמי יכול להגביר יעילות השימוש בהם. זה יכול לעבוד רק כאשר כל ה-VMs על המכונה שייכים לאותו הארגון / יש ביניהם הסכמה. T3/T2 instances ב-EC2 הם VMs שעובדים על עיקרון דומה: ב"חוזה" שלנו רשום שה-instance יכול לעבוד ב-burst ולקבל יותר משאבים - אך עדיין יש פה עבודה לפי חוזה, ולא אופטימיזציה גלובלית של המשאבים על המכונה (מכיוון שה-VMs שייכים לארגונים שונים).



הכרה חברתית

יהיה לא נכון להתעלם מהתשואה החברתית של השימוש בקוברנטיס. מי שמשתמש היום בקוברנטיס, ובמיוחד בפרודקשן - נתפס כמתקדם טכנולוגית, וכחדשן. ההכרה החברתית הזו - איננה מבוטלת.

כמובן שזו הכרה זמנית, שרלוונטית רק שנים בודדות קדימה. ביום לא רחוק, שימוש בקוברנטיס ייחשב מפגר ומיושן, ועל מנת להיתפס כמתקדמים / חדשנים - יהיה עלינו לאמץ טכנולוגיה אחרת.

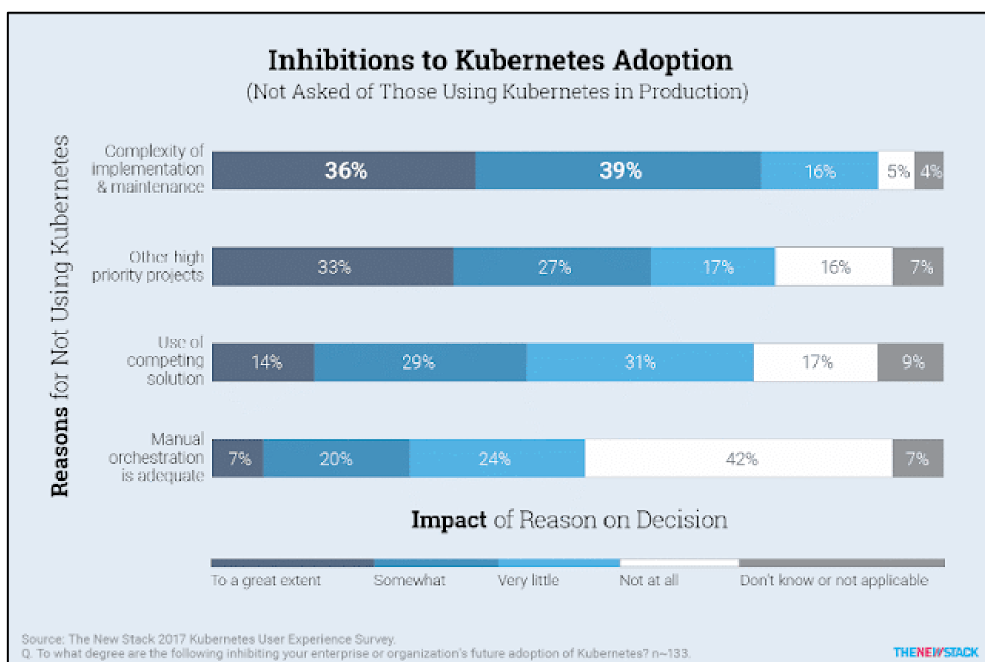
יש לי תחושה שפעמים רבות, ההכרה החברתית היא שיקול חזק לא פחות משיקולי היעילות - בבחירה בקוברנטיס. טבע האדם.

מחירים

כמובן שיש לכל הטוב הזה גם מחירים:

- קוברנטיס היא טכנולוגיה חדשה שיש ללמוד - וכמות / מאמץ הלמידה הנדרש הוא לרוב גבוה ממה שאנשים מצפים לו.
- התסריטים הפשוטים על גבי קוברנטיס נראים די פשוטים ואוטומטים. כאשר נכנסת לתמונה גם אבטחה, הגדרות רשת, ותעדוף בין מיקרו-שירות אחד על האחר - הדברים הופכים למורכבים יותר! Troubleshooting - עשוי להיות גם דבר לא פשוט, מכיוון ש"מתחת למכסה המנוע" של קוברנטיס - יש מנגנונים רבים.

- ברוב המקרים נרצה להריץ את קוברנטיס על שירות ענן, ולכן נידרש עדיין לשמר מידה של מומחיות כפולה בשני השירותים: לשירות הענן ולקוברנטיס יש שירותים חופפים כמו Auto-Scaling, הרשאות ו-Service Discovery (בד"כ: DNS).
- הטכנולוגיה אמנם לא ממש "צעירה", והיא בהחלט מוכחת ב-Production - אך עדיין בסיסי הידע והקהילה התומכת עדיין לא גדולה כמו פתרונות ענן מסחריים אחרים. יש הרבה מאוד אינטגרציות, אך מעט פחות תיעוד איכותי וקל להבנה.
- כמו פעמים רבות בשימוש ב-Open Source - אין תמיכה מוסדרת. יש קהילה משמעותית ופעילה - אבל עדיין הדרך לפתרון בעיות עשויה להיות קשה יותר מהתבססות על פתרון מסחרי.
- גם בשימוש ב"קוברנטיס מנוהל" (EKS, AKS, ו-GKE), החלק המנוהל הוא החלק הקטן, והשאר - באחריותנו.
- האם החיסכון הצפוי מניהול משאבים יעיל יותר, יצדיק במקרה שלכם שימוש בסביבה שדורשת מכם יותר תפעול והבנה?
- במקרה של ניהול מאות או אלפי שרתים - קרוב לוודאי שזה ישתלם.
- שימוש בקוברנטיס עשוי לפשט את סביבת התפעול, וה Deployment Pipeline. ההשקעה הנדרשת היא מיידית - בעוד התשואה עשויה להגיע רק לאחר זמן ניכר, כאשר היישום הספציפי באמת הגיע לבגרות.
- במקרים לא מעטים, ארגונים נקלעים לשרשרת של החלטות שנגזרות מצו האופנה ובניגוד לאינטרס הישיר שלהם: עוברים למיקרו-שירותים כי "כך כולם עושים" / "סיפורי הצלחה" שנשמעים, משם נגררים לקוברנטיס - כי יש להם הרבה מאוד שרתים לנהל, שכבר נהיה דיי יקר. לו היינו עושים שיקולי עלות/תועלת מול המצב הסופי - כנראה שהרבה פעמים היה נכון לחלק את המערכת למודולים פשוטים, או להשתמש במיקרו-שירותים גדולים ("midi-services") - וכך לשלוט טוב יותר בעלויות והמורכבויות האחרות.

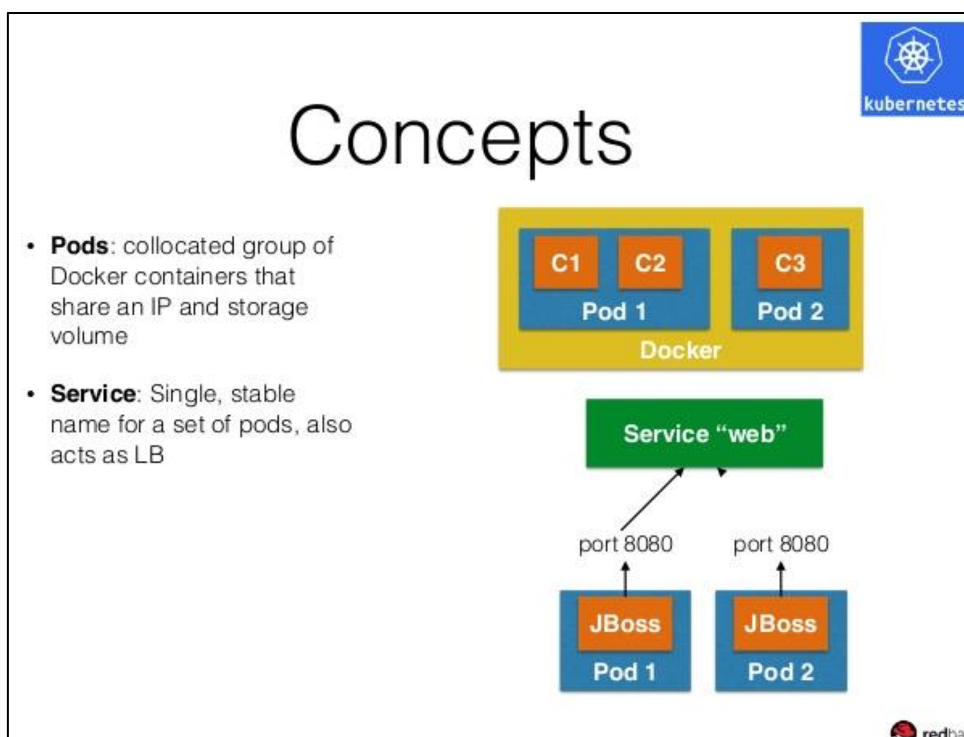


קוברנטיס בפועל

דיברנו עד עכשיו על עקרונות ברמה הפשטה גבוה. בואו ניגש לרמה טכנית קצת יותר מוחשית.

לצורך הדיון, נניח שהעברנו כבר את כל השירותים שלנו לעבוד על-גבי Docker וגם התקנו כבר Cluster של קוברנטיס. זה תהליך לא פשוט, שכולל מעבר על כמה משוכות טכניות לא קלות - אך נניח שהוא נגמר. בכדי להבין קוברנטיס חשוב יותר להבין מה יקרה אחרי ההתקנה, מאשר להבין את ההתקנה עצמה.

לצורך הדיון, נניח שאנו עובדים על AWS ו-EKS ואמזון מנהלים עבורנו את ה-Masters nodes. ה- Worker nodes שלנו נמצאים ב-Auto-Scaling Group - מה שאומר שאמזון תנהל עבורנו את ה-nodes מבחינת עומס (תוסיף ותוריד מכונות ע״פ הצורך) והחלפת שרתים שכשלו. זה חשוב! אנחנו גם משתמשים ב-ECR (קרי Container Registry מנוהל) , ואנו משתמשים בכל האינטגרציות האפשריות של קוברנטיס לענן של אמזון (VPC, IAM, ELB, וכו׳). התצורה מקונפגת ועובדת היטב - ונותר רק להשתמש בה. אנחנו רק רוצים ״לזרוק״ קונטיינרים של השירותים שלנו - ולתת ל״קסם״ לפעול מעצמו. רק לומר בפשטות מה אנחנו רוצים - ולתת לקוברנטיס לדאוג לכל השאר!



יצירת Pod

הפעולה הבסיסית ביותר היא הרצה של Container. בקוברנטיס היחידה האטומית הקטנה ביותר שניתן להריץ נקראת Pod והיא מכילה Container אחד או יותר. כרגע - נתמקד ב-Pod עם Container יחיד, זה יהיה המצב ברבים מהמקרים.

אני מניח שאנחנו מבינים מהו Container (אם לא - שווה לחזור צעד אחורה, ולהבין. למשל: הפוסט שלי בנושא), ויש לנו כבר Image שאנו רוצים להריץ ב-ECR. בכדי להריץ Container, עלינו לעדכן את קוברנטיס ב-manifest file המתאר Pod חדש מצביע ל-container image. קוברנטיס ירשום את ה-Pod ויתזמן אותו לרוץ על אחד מה nodes שזמינים לו. כאשר ה-node מקבל הוראה להריץ Pod עליו להוריד את ה-container image - אם אין לו אותו כבר. כל node מחזיק עותקים עצמאיים של ה- container images משיקולים של high availability.

הנה קובץ ה-manifest שהרכבנו:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    env: dev
    version: v1
spec:
  containers:
  - name: hello-world-ctr
    image: hello-world:latest
    ports:
    - containerPort: 8080
      protocol: TCP
```

קובץ ה-manifest בקוברנטיס מורכב מ-4 חלקים סטנדרטיים:

1. **גרסת ה-API**
   1. הפורמט הוא לרוב <api group>/<version> אבל כמה הפקודות הבסיסיות ביותר בקוברנטיס נמצאות תחת API Group שנקרא core - ולא צריך לציין אותו.
   2. ה-API של קוברנטיס נמצא (בעת כתיבת המאמר) בגרסה 1.13 - אז למה גרסה 1? ניהול הגרסאות בקוברנטיס הוא ברזולוציה של משאב. הקריאה לייצור pod היא עדיין בגרסה 1 (כמו כמעט כל ה-APIs. בעת כתיבת המאמר אין עדיין גרסה v2 לשום API, מלבד v2alpha או v2beta - כלומר גרסאות v2 שעדיין אינן GA).
2. **סוג (kind)** - הצהרה על סוג האובייקט המדובר. במקרה שלנו: Pod.
3. **metadata** - הכולל שם ו-labels שיעזרו לנו לזהות את ה-pod שיצרנו.
   1. ה-labels הם פשוט זוגות key/value שאנחנו בוחרים. הם חשובים מאוד לצורך ניהול Cattle של אובייקטים, והם בלב העבודה בקוברנטיס.

4. **spec** - החלק המכיל הגדרות ספציפיות של המשאב שהוגדר כ-"Type".
   1. name - השם שניתן ל-container בתוך ה-Pod, וצריך להיות ייחודי. במקרה של container יחיד בתוך ה-pod - אין בעיה כזו.
   2. image - כמו בפקודת docker run...
   3. ports - ה-port שיהיה published. TCP הוא ערך ברירת-המחדל, אך הוספתי אותו בכדי לעשות את ה-Yaml לקריא יותר.

תזכורת קצרה על Yaml:

פה ייתכן וצריך לעצור שנייה ולחדד כמה מחוקי-הפורמט של Yaml. ייתכן ונדמה לכם ש-Yaml הוא פורמט פשוט יותר מ-JSON - אבל זה ממש לא נכון. זה פורמט "נקי לעין" - אבל מעט מורכב. רשימה ב-Yaml נראית כך:

```
mylist:
   - 100
   - 200
```

"mylist" הוא ה-key, והערך שלו הוא רשימה של הערכים 100 ו-200. כל האיברים שלפניהם הסימן " - " ובעלי עימוד זהה - הם חברים ברשימה. סימן ה-Tab ברוב ה-Editors מתתרגם ל-2 או 4 רווחים. ב-Yaml הוא שקול לרווח אחד, ולכן שימוש בו הוא מקור לבעיות ובלבולים. הימנעו משימוש ב-Tab בתוך קבצי Yaml!

המבנה הבא, שגם מופיע ב-manifest (ועשוי לבלבל) הוא בעצם רשימה של Maps:

```
channels:
  - name: '#mychannel'
    password: ''
  - name: '#myprivatechannel'
    password: 'mypassword'
```

"channel" הוא המפתח הראשי, הכולל רשימה. כאשר יש "מפתח: ערך" מתחת ל"מפתח: ערך" באותו העימוד - משמע שמדובר ב-Map. כלומר, המפתח "channel" מחזיק ברשימה של שני איברים, כל אחד מהם הוא מפה הכוללת שני מפתחות "name" ו-"password".

אם נחזור לדוגמה של הגדרת ה-container ב-manifest למעלה, בעצם מדובר במפתח "containers" המכיל רשימה של איבר אחד. בתוך הרשימה יש מפה עם 3 מפתחות ("image", "name" ו-"ports") כאשר המפתח האחרון "ports" מכיל רשימה עם ערך יחיד, ובה מפה בעלת 2 entries.

הנה [מדריך המתאר את ה-artifacts הבסיסיים ב-Yaml] ממנו לקחתי את הדוגמאות. חשוב להזכיר שיש עוד כמה וכמה artifacts ב-Yaml - אם כי כנראה שלא נזדקק להם בזמן הקרוב.

עכשיו כשיש לנו manifest, אנחנו יכולים להריץ את ה-Pod:

```
$ kubectl apply -f my-manifest-file.yml
```

kubectl הוא כלי ה-command line של קוברנטיס. פקודות מסוימות בו יזכירו לכם את ה-command line של docker. במקרה הזה אנו מורים לקוברנטיס להחיל קונפיגורציה. הפרמטר -f מציין שאנו מספקים שם של קובץ.

תוך כמה עשרות שניות, לכל היות, ה-pod שהגדרנו אמור כבר לרוץ על אחד ה-nodes של ה-cluster של קוברנטיס.

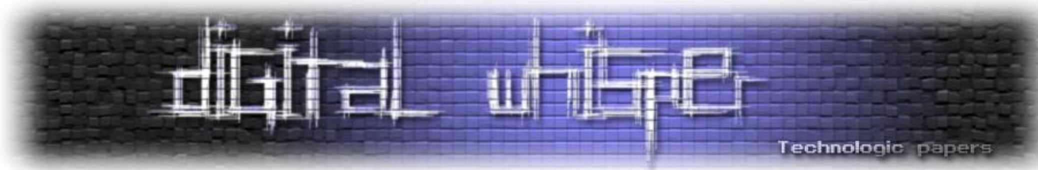
אנו יכולים לבדוק אלו Pods רצים בעזרת הפקודה הבאה:

```
$ kubectl get pods
```

עמודה חשובה שמוצגת כתוצאה, היא עמודת הסטטוס - המציגה את הסטטוס הנוכחי של ה-pod. אמנם יש [רשימה סגורה של מצבים] בו עשוי להיות pod, אולי עדיין הסטטוס המדווח יכול להיות שונה. למשל: הסטטוס ContainerCreating יופיע בזמן שה-docker image יורד ל-node. זה מצב נפוץ - אך לא מתועד היטב. את הסטטוס ניתן למצוא בעיקר... ב[קוד המקור של קוברנטיס].

הפקודה הבאה ([וריאציות]), בדומה לפקודת ה-Docker המקבילה - תציג את הלוגים של ה-Container ב-Pod:

```
$ kubectl logs my-pod
```



אם ב-Pod יש יותר מ-2 containers (מצב שלא אכסה במאמר), הפקודה תציג לוגים של ה-container הראשון שהוגדר ב-manifest. אפשר לציין את שם ה-container כפי שצוין ב-manifest - וכך להגיע ל-container נתון בתוך ה-Pod מרובה ה-containers.

עבור תקלות יותר בסיסיות (למשל: ה-pod תקוע על מצב ContainerCreating וכנראה שה-node לא מצליח להוריד את ה-container image) - כדאי להשתמש בפקודה:

```
$ kubectl describe pods my-pod
```

התוצאה תהיה סטטוס מפורט שיכיל את הפרטים העיקריים מתוך ה-manifest, רשימה של conditions של ה-pod, ורשימת כל אירועי-המערכת שעברו על ה-pod מרגע שהורנו על יצירתו. הנה דוגמה להפעלה הפקודה (מקור):

```
Name:  nginx-deployment-1006230814-6winp
Node:  kubernetes-node-wul5/10.240.0.9
Start Time: Thu, 24 Mar 2016 01:39:49 +0000
...
Status:  Running
IP:  10.244.0.6
Controllers: ReplicaSet/nginx-deployment-1006230814
Containers:
  nginx:
    Container ID: docker://90315cc9f513c750f244a355eb1149
    Image:  nginx
    Image ID:  docker://6f623fa05180298c351cce53963707
    Port:  80/TCP
    Limits:
      cpu: 500m
      memory: 128Mi
    State:  Running
      Started:  Thu, 24 Mar 2016 01:39:51 +0000
    Ready:  True
    Restart Count: 0
    Environment:       <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-5kdvl (ro)
Conditions:
  Type          Status
  Initialized   True
  Ready         True
  PodScheduled  True
Volumes:
  default-token-4bcbi:
    Type: Secret (a volume populated by a Secret)
    SecretName: default-token-4bcbi
    Optional:   false
...
Events:
  FirstSeen LastSeen Count From     SubobjectPath  Type  Reason  Message
  --------- -------- ----- ----     -------------  -------- ------  -------
  54s  54s  1 {default-scheduler }      Normal  Scheduled Successfully assigned nginx-deployment-1006230814-6winp to kubernetes-node-wul5
  54s  54s  1 {kubelet kubernetes-node-wul5} spec.containers{nginx} Normal  Pulling  pulling image "nginx"
  53s  53s  1 {kubelet kubernetes-node-wul5} spec.containers{nginx} Normal  Pulled  Successfully pulled image "nginx"
  53s  53s  1 {kubelet kubernetes-node-wul5} spec.containers{nginx} Normal  Created  Created container with docker id 90315cc9f513
  53s  53s  1 {kubelet kubernetes-node-wul5} spec.containers{nginx} Normal  Started  Started container with docker id 90315cc9f513
```

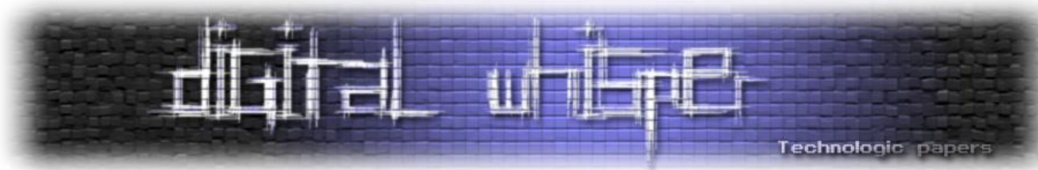
ניתן גם, בדומה ל-Docker, לגשת ישירות ל-console של ה-container שרץ ב-pod שלנו בעזרת הפקודה:

```
$ kubectl exec my-pod -c hello-world-ctr -it -- bash
```

במקרה הזה ציינתי את שם ה-container, אם כי לא הייתי חייב. נראה לי שזה מספיק, לבינתיים.

בואו נסגור את העניינים:

```
$ kubectl delete -f my-manifest-file.yml
```



הפעולה הזו עלולה להיראות מוזרה ברגע ראשון. הסרנו את הקונפיגורציה - ולכן גם ה-Pod ייסגר?

לשימוש בקוברנטיס יש שתי גישות עיקריות:

- גישה אימפרטיבית - בה מורים לקוברנטיס איזה שינוי לבצע: להוסיף משאב, לשנות משאב, או להוריד משאב.
- פקודות כגון kube ctl create או kubectl replace הן בבסיס הגישה האימפרטיבית.
- גישה דקלרטיבית - בה מורים לקוברנטיס מה המצב הרצוי - והוא יגיע עליו בעצמו.
- פקודת kubectl apply - היא בבסיס הגישה הדקלרטיבית. אפשר להגדיר כמעט הכל, רק באמצעותה.
- החלת patch על גבי קונפיגורציה קיימת הוא משהו באמצע: זו פקודה דקלרטיבית, אך מעט חורגת מה lifecycle המסודר של הגישה הדקלרטיבית הקלאסית. סוג של תרגיל נינג'ה.

כמובן שהגישה הדקלרטיבית נחשבת קלה יותר לשימוש ולתחזוקה לאורך זמן - והיא הגישה הנפוצה והשלטת.

מכיוון שאין לנו הגדרה אלו Pods קיימים סה"כ במערכת - לא יכולנו לעדכן את המערכת בהגדרה הכוללת - ולכן הסרנו את המניפסט.

סיכום

ניסינו להגדיר בקצרה, ובצורה עניינית מהם היתרונות הצפויים לנו משימוש בקוברנטיס - מול הצפה של החסרונות והעלויות. לא פעם קראתי מאמרים שמתארים את קוברנטיס כמעבר מהובלת מסע על גבי פרד - לנסיעה ברכב שטח יעיל! אני חושב שההנחה הסמויה ברוב התיאורים הללו היא שהקוראים עדיין עובדים על גבי מערכות On-Premises ללא טכנולוגיות ענן. מעבר משם לקוברנטיס - היא באמת התקדמות אדירה.

המעבר משימוש ב"ענן של מכונות וירטואליות" לקוברנטיס - הוא פחות דרמטי. קוברנטיס תספק בעיקר יעילות גבוהה יותר בהרצת מיקרו-שירותים, וסביבה אחידה וקוהרנטית יותר למגוון פעולות ה-deployment pipeline. עדיין, נראה שקוברנטיס היא סביבה מתקדמת יותר - ואין סיבה שלא תהפוך לאופציה הנפוצה והמקובלת, בתוך מספר שנים. אחרי ההקדמה, עברנו לתהליך יצירה של Pod בסיסי, ובדרך עצרנו בכמה נקודות בכדי להדגים כיצד התהליך "מרגיש" בפועל.

חשוב לציין שה-Pod שהגדרנו הוא עצמאי ו"חסר גיבוי" - מה שנקרא "naked pod". אם naked pod כשל מסיבה כלשהי (הקוד שהוא מריץ קרס, או ה-node שעליו הוא רץ קרס/נסגר) - הוא לא יתוזמן לרוץ מחדש. מנגנון ה-auto-healing של קוברנטיס שייך לאובייקט / אבסטרקציה גבוהה יותר בשם ReplicaSet. אבסטרקציה מעט יותר גבוהה, שבה בדרך כלל משתמשים - נקראת Deployment. כיסינו דיי הרבה למאמר אחד. הנושאים הללו מצדיקים מאמר משלהם.

שיהיה בהצלחה!

באגים נסתרים

מאת עידו קנר

הקדמה

בעקבות דיונים ארוכים אשר נכחתי בהם מספר פעמים, החלטתי להעלות למאמר את דעתי בנושא שאני מוצא כי יש בו מחסור רב בהבנת המשמעות של הכלים בזמן פיתוח תוכנה.

למרות כי על פניו, נראה כי כל מה שצריך הוא פוסט בבלוג, חשבתי כי הנושא חשוב מספיק בשביל להיות נגיש ליותר אנשים ועל כן החלטתי לפרסמו כמאמר במגזין.

במאמר קצר זה אנסה להציג מדוע לדעתי יש בעיות בתפיסה בה ביצוע בדיקות על תוכנה נחשבות להעדר באגים, ומה המשמעות של אמונה זו מבחינת בעיות אשר כן יכולות להגיע לתוכנה.

חשוב לי להדגיש כי זו הראיה שלי בלבד, המגיעה מניסיון שלי ושל מכרים בתחום התוכנה, וכתיבת מאמר זה אינו מגיע לתקוף דעות המנוגדות לשלי, אלא להציג כשלים שונים אשר אני חוויתי בעת אותם הדיונים.

תפיסת עולם

באפריל 2018 העברתי הרצאה למספר ארגונים שונים אודות מסדי נתונים רלציונים המתוחזקים כקוד פתוח. אחת השאלות הראשונות אשר נשאלתי היתה "איך זה לא מסוכן לדעת מה מנגנון האבטחה של מסד הנתונים, כאשר הקוד שלו גלוי?"

התשובה שלי הייתה מאוד פשוטה: "האם הקוד של iOS חשוף לעולם?" נעניתי בשלילה. האם עדיין יש פריצות אשר מאפשרות לקבל גישת ניהול על המכשיר? התשובה היתה חיובית. כך שבעצם Security through Obscurity יכול להקשות, אבל לא למנוע מגילוי בעיות.

מימוש טוב, ללא באגים ידועים גורמים לכך שהאלגוריתמים, ללא קשר אם חושפים את הקוד שלהם או לא, מקשים מאוד על פריצה כלשהי. כמובן שאין זה אומר כי אין חולשות, אלא כי הן אינן מוכרות לציבור.

כלומר החולשה היא לא הידע כיצד מומש האלגוריתם, אלא בניצול של פער הקיים במימוש, בין אם מדובר במימוש כושל או במימוש למנגנון כושל (כשל לוגי בדרך כלל). ניתן לעשות למימוש האלגוריתם Reverse Engineering גם אם אין את קוד המקור, אך ניתן למצוא בעיות גם כאשר קוראים את קוד המקור.

הנה הסבר מוחשי יותר: כאשר יודעים כי יש מימוש קוד המממש את x509 למשל, כל עוד הקוד של x509 ללא באגים ידועים, וכל עוד אין כשל לוגי בפרוטוקול, יהיו חייבים לשנות את הקוד בשביל שיהיו בעיות אבטחה. על מנת לנסות לנסות ולמצוא בעיות, יהיו חייבים לחקור אותו הרבה מאוד זמן בשביל למצוא מה אולי מומש לא טוב ונכון. במידה ולא נמצאה חולשה כלשהי במימוש, לא תהיה פריצה אליו, אולי ניצול של דברים מסביב, אבל לא של המימוש שבוצע ל-x509.

כאשר מדובר במערך בדיקות, אני מגלה כי יש תפיסה שגויה בהרבה פעמים אודות המשמעות של בדיקות.

הרעיון של ביצוע בדיקות אינו משקף הרבה מאוד דברים להם יש יחס כאשר מדברים על אותו מערך בדיקות, ואני חושב כי הגיע הזמן לנסות לפקס ולמקד קצת מעבר לכך מה התפקיד של אותו מערך בדיקות עצמו. זאת לטובת כך שמערכי הבדיקות לא יספקו תחושות שגויות באשר למה שהן מספקות.

כאשר אני מדבר על מערך בדיקות תוכנה, אני מדבר **רק** על שלושת הבדיקות הבאות:

- יחידות בדיקה (unit test): בדיקת פונקציונליות של פונקציה/מתודה מדוייקת, לתת אפשרות מסוימת.
- בדיקת השתלבות (integration test): בדיקת שילוב של מספר פונקציות לקבל פונקציונליות רצויה.
- בדיקות קצה לקצה (end to end/e2e): בדיקת תוצר רצוי כאשר מבצעים פעולות מלאות.

למרות שישנם סוגים נוספים של מערכי בדיקות, אינני הולך להתייחס אל אותם המערכים...

תפקיד הבדיקות

אז מדוע בכלל לבצע בדיקות תוכנה? התשובה לכך תלויה בסוג הבדיקה. למשל בדיקות של unit test, מבצעות בדיקה כי קלט או פלט מסוימים עושים את העבודה כמצופה בנקודה מסוימת. כלומר נקודת לוגיקה אחת בתוך פונקציה אחת בלבד נבדקת, ולא בדיקה של כלל הפונקציה.

כאשר משנים פונקציה כלשהי, רוצים בנוסף לבדיקה גם לראות שלא נשבר משהו שעבד בעבר (או אולי כן במידה וצריך), והתפקיד של unit test הוא לספק את עצם הוודאות כי אנו יודעים שכל מה שרצינו לקבל מתקבל, כל עוד אנו יודעים מה אנו רוצים לקבל.

זו הסיבה גם כי בודקים נקודת לוגיקה אחת פשוטה בפונקציה ולא את כולה.

כאשר מדובר ב-integration test, אנו רוצים (לצורך ההסבר) לדעת אם אנו מצליחים לקבל דוא"ל, לדעת כי הדוא"ל קיים במערכת ולשלוח הודעה בהצלחה (או לקבל כישלון נכון). אנו משלבים מספר פונקציות שאנו יודעים מה התוצר שלהן אמור להיות בכל שלב, ובכך יודעים כי לוגיקה מסוימת מתקיימת לפי הצורך. הבדיקה היא רק על התנהגות מאוד מוגדרת של דוא"ל, אך לא מעבר. זה תוצר של מספר פונקציות, אשר אחראיות לכך. למשל גם פונקציות המספקות פרסרים, פונקציות היוצרות פרוטוקול SMTP וכו', עד שמגיעים למצב בו מתקבלת הודעת דוא"ל ומפורשת בהצלחה.

כאשר מדובר בבדיקות e2e, אנו רוצים לבדוק כי אנו מקבלים מסך login, וכאשר מזינים משתמש וסיסמה נכונים, עוברים לדף הבא. בדיקות e2e למעשה מקבלות תוצר של מספר לוגיקות אשר יוצרות תוצר אותם המשתמשים מקבלים.

כלומר הבדיקה היא קבלה של מסך מסוים, האם יש שם שדות שמחפשים אותן, האם יש כפתור לכניסה, האם לחיצה עם שדות מלאים מספקים את מה שציפינו לו, כגון כניסה בהצלחה או הזדהות שגויה. בדיקת e2e היא למעשה בדיקה מה החוויה מבחינת סדר פעולות אשר המשתמש נדרש לעשות והאם פעולות אלו עוברות בהצלחה.

כעת, לאחר שעברנו יישור קו עם המונחים וכיצד אני מתייחס אל אותם המונחים, ניתן להמשיך.

מדוע יש בעיות עם בדיקות?

לפני מספר שנים הייתי בהרצאה על כמה חשוב לקבל מערך בדיקות (התשובה: כי הוא למעשה 'התנ"ך' של המערכת). במידה ויש כיסוי מאוד מסיבי, סביר להניח כי אפשר להבין דברים רבים על המערכת, אבל מערך בדיקות אינו מראה את איכות המערכת, ובוודאי שלא מצביע על העדר בעיות ובאגים.

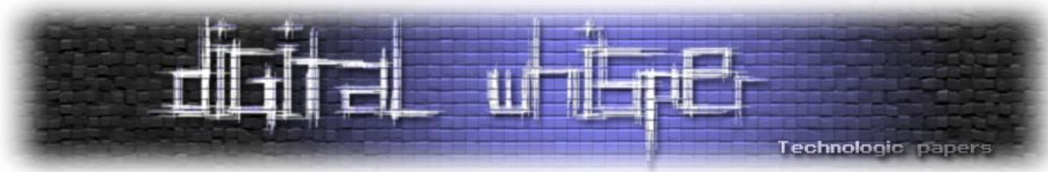
יותר מכך, יכול להיות שכלל המערך בודק דברים שגויים, וככזה הוא מקשה על הבנה כיצד המערכת בנויה, וזאת - במקום לספק את היכולת להבין נכון את המערכת.

ואם לא די כאן, כבר נתקלתי במערך בדיקות אשר הקוד היה מאוד קריפטי ורק מי שכתב את הבדיקות הבין כיצד הוא מבצע את הבדיקות בפועל. כלומר הבנתי לפי השם של הבדיקה מה היא בודקת, אבל לא היה ברור בכלל כיצד זה אמור להגיע לידי ביטוי, אך איכשהו הבדיקה עבדה, ולי לא היתה דרך להבין אם היא באמת בודקת את מה שרצה אותו מפתח או לא.

בדיקות תוכנה הן למעשה עוד עטיפת קוד, אשר מחביאות בעצמן בעיות נוספות. הבדיקות מציגות כי מה שנבדק עובד, אך בצורה אשר רק מי כתב אותם רצה. וזאת כל עוד המערך בדיקות אינו מכיל באגים ובעיות אחרות בעצמן.

בדיקה לא נכונה, או הבנה לא נכונה של תפקיד הפונקציה, יכולה לתת תוצאה נכונה מסיבות לא נכונות. ובכך הבדיקות למעשה אינן משקפות את תקינות המערכת. כלומר בדיקות יכולות גם לספק false positive או true negative. הסיבות לכך הם שיש מספר רב של נעלמים שאיננו מודעים להם.

למשל עם עורך טקסט, כאשר הוא מקבל תו null, מה קורה? נגיד והתו הראשון הוא null, האם העורך יגיב בצורה זהה כמו התו שני? מה קורה אם רק בתו ה-10,000,005 יקרה משהו? סדרה של כמה null עלולה לגרום לערוך הטקסט להתנהג בצורה לא צפויה? פשוט לא ניתן לחזות כל דבר. זו הסיבה כי באגים כדוגמת הבאג הזה, לפעמים מתגלים רק במצבים קיצוניים בלבד.



עוד הדגמה היא, יצירת מערכת שתוכל להתמודד עם Yottabyte של מידע. בזמן כתיבת שורות אלו, אני חושב כי מדובר במשהו תיאורטי בלבד, אבל כיצד המערכת תתמודד עם גודל מידע אסטרונומי שכזה?

הבעיה על Yottabyte נשמעת כנושא תיאורטי בזמן כתיבת שורות אלו, אך בפודקאסט ששמעתי לפני הרבה מאוד שנים, דובר על באג אשר היה במימוש של XFree ששם ה-Buffer לא בדק חריגה של גודל מידע מסוים אשר בזמן הכתיבה של המערכת, זה היה גודל תיאורטי בלבד, ולא היו בכלל מחשבים אשר הכילו את אותו הגודל, וכאשר הטכנולוגיה סיפקה גודל שכזה, התגלה באג שתורגם בסופו של דבר ל-root elevation עם buffer overflow ומספר בעיות נוספות.

כאשר אני מציג בעיות כאלו, התשובות המתקבלות הרבה פעמים הם: "אז אתה אומר כי בדיקות תוכנה אינן צריכות להתקיים?". ובכן, התשובה היא שיש להן הרבה מקום, אבל חובה להבין בדיוק את תפקידן, והוכחה של היעדרות בעיות אינן אחת מהן.

התשובה היא לא קיצונית, אין כאן מצב של "הכל או כלום", אין מקום לגישה שכזו, היות ויש מקום למערך הבדיקות, אך חשוב להבין מה התפקיד שלהן ואיפה הוא נותן תחושה שגויה של ביטחון.

כמה בדיקות צריך בשביל להחליף נורה?

אנחנו מכירים את זה: מגיע מנכ"ל של חברה, עומד על במה, מדגים את המוצר החדש של החברה לקהל גדול, גם אנשי תקשורת, כולם בהתלהבות של ההשקה, אותו מדגים מדבר כי על כל 10,000 שורות קוד של המערכת יש כ-100,000 בדיקות, וזה המוצר עם הכי הרבה בקרת איכות שאי פעם בוצע בחברה, ו... אז תוך כדי הצגה של המערכת - היא קרסה למדגים.

אם יש בדיקות רבות כל כך, איך עדיין יש באגים? או לחילופין:

מדוע בדיקות אינן מוכיחות העדר באגים?

בעולם אבטחת המידע יש כלי הנקרא fuzzer. התפקיד של הכלי הוא לייצר מידע רנדומלי בתבניות שונות על מנת למצוא בעיות בתוכנות. זה למעשה אמצעי בדיקות אוטומטי אשר תפקידו למצוא נקודות כשל בתוכנה שלא סביר לחשוב עליהן לבד.

כלומר, אני כבן אדם מתקשה הרבה פעמים לחשוב על הרבה מאוד תבניות למערכת שאני בודק, ונוסחאות אוטומטיות שהמחשב מבצע עבורי מוסיפות עוד מידע שניתן לבדוק אותו במערכת אותה אני מפתח.

כאשר, אני מתכנן ממשק משתמש, אשר ניתן להוסיף רשימה באמצעות פסיקים של כתובות IP. מגיע משתמש ולא קורא את ההוראות ושם לי רווח או שורה חדשה פר כתובת IP, מה עכשיו? כיצד המערכת שלי תגיב? ומה אם הוא הכניס לי מפריד אחר? האם אני יכול לנחש כל סוג מפריד שהוא?

היות ואין לי יכולת (או זמן) לבדוק כל סוג קלט אפשרי, בכל מבנה אפשרי, אין לי יכולת לכתוב בדיקות אשר יצליחו לבדוק כל מבנה אפשרי וסדר אפשרי של כל תו בכל אינדקס אפשרי, ארצה להשתמש ב-fuzzer על מנת לנסות ולכסות מקרים רבים יותר ככל הניתן.

אך גם כלים כדוגמת fuzzer אינם מסוגלים לייצר את כל האופציות. הם יכולים ליצור dataset מסוים ולהגיע לקלטים רבים יותר מבני אדם, אך ה-fuzzer עדיין לא יכול לספק מענה הולם לכל מקרה אפשרי. השימוש בבדיקות חייב להיות שונה בתפיסה שלו ובשימוש שלו. ההבנה כי באגים יכולים להתקיים, לא משנה מה כמות הבדיקות, היא חשובה מאוד בנושא.

סיבה לכך כי גישה הזו של "יש לי באגים במערכת", אומרת כי יצרן המערכת אינו נתפס בהפתעה, ובכך בניהול הבעיה הוא מגיע מוכן, במידה והיא פשוטה או במידה והיא חמורה עדיין יש מקום רב יותר להתמודדות עם הבעיה ותיקון טוב יותר שלה.

גם בדיקה היא קוד

"Quis custodiet ipsos custodes?" הוא משפט לטיני השואל "מי ישמור על השומרים?" כיצד אפשר להבטיח כי בדיקה אינה מכילה באגים בפני עצמה? הרי גם בדיקה היא למעשה קטע קוד, ויש סיכוי כי גם לה יש באגים. ולכן, בדיקות צריכות להתבצע, אך יש לקחת אותן בערבון מוגבל.

האם כאשר בדיקה נכשלת, זה בהכרח אומר כי הקוד שאותה הבדיקה בודקת שגוי? האם כאשר בדיקה עוברת, זה בהכרח אומר כי הקוד שאותה הבדיקה בודקת תקין?

התשובה לשתי השאלות הללו היא כמובן "לא".

אם עד עכשיו דיברתי באוויר, ארצה להדגים לכם את הבעיה, ואשתמש בקוד Go לשם כך:

```
func IsOne(input string) bool {
  return input == “1”
end
```

שורת לוגיקה אחת, בודקת האם מחרוזת מכילה רק את הספרה "1" כמחרוזת בתוכה, נכון? אצור לה כעת בדיקות:

```
func TestIsOneWithZero(t *testing.T) {
   result := IsOne(“0”)
   if result == true {
     t.Error(“`result` expected to be `false`, but got `true`”)
   }
}

func TestIsOneWithOne(t *testing.T) {
```

```
  result := IsOne("1")
  if result == false {
    t.Error("`result` expected to be `true`, but got `false`")
  }
```

האם הבדיקות אשר כתבתי למעלה בודקות את כל המצבים האפשריים עבור שורת לוגיקה אחת של IsOne?

תחושת הבטן אומרת לי שכן, אבל הניסיון אומר לי שלא!

אתחיל בהתחכמות, ואגיד כי אני בודק רק את התו "1" באסקי, אך למשל מה עם 'Mathematical bold digit one'

אני יודע, אפשר להגדיר כי התפקיד של אותה הפונקציה היא להחזיר רק אם הערך אסקי של "1" מתקיים. במידה ואגדיר כי רק הערך הזה מתקיים עבורי כערך של הספרה "אחד", פתרתי את הבעיה הזו.

מה קורה כאשר אני מקבל מחרוזת של מספר תווים שכולם הספרה "1"? האם יש הגדרה כמה פעמים מופיעה הספרה הזו? האם אני צריך להגדיר כי הופעה חד פעמית בלבד נדרשת?

לאחר הגדרות אלו, זה אומר כי אני צריך לכתוב עוד בדיקות, ולהבטיח כי כל הגדרה אשר אני רוצה כי לא תתקיים, בהכרח לא מתקיימת.

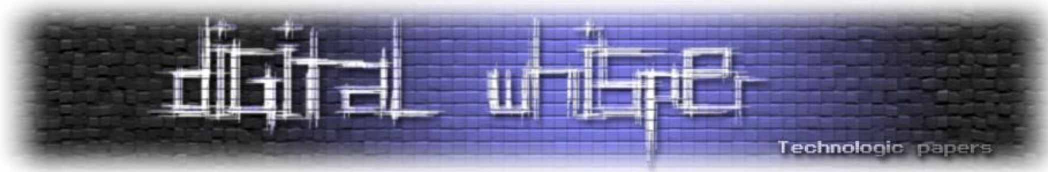
ומה אם פתאום גיליתי תו חדש בתקן Unicode שרק פורסם אשר מכיל את הספרה "1" בצורה אחרת, כיצד הקוד שלי יתמודד אז? במצב כזה, תמיד אני במירוץ אחר מציאת הגדרות מאוד מדויקות מה נכון ולהבטיח כי כל דבר אחר יחזיר לי "לא נכון".

והנה באג במערכת: מסתבר כי כתבתי את הפונקציה IsOne בצורה מסובכת אשר מרגישה כטובה. מה הכוונה?

```
func IsOne(input string) bool {
  return len(input) == 1 &&
         rune(input[0]) == 0x31
end
```

אני בודק האם האורך הוא "1", כי אם לא, אין טעם לבדוק את השאר, נכון?

כעת, על מנת להבטיח כי קיבלתי באמת את הספרה "1", אני בודק האם מה שנמצא לי בתא "0" במחרוזת זה בעצם הערך האסקי של "1". בשורה של rune, בעצם אני מחביא היתכנות לבאג. למי שאינו יודע, ב-Go, המחרוזת מכילה כברירת המחדל UTF-8, מה שאומר שתו אחד יכול להיות בין בית בודד עד לארבעה בתים. אני בודק אורך של תו, ולא אורך של בית. אני בודק עם rune האם התו (אשר יכול להיות להיות עד 4 בתים) הוא 0x31. עד כאן הכל תקין. אבל האם ידעתם כי Unicode כתקן מחזיק בתוכו מבנה ביטים מסוים אשר יכול להבטיח כי זה אינו בית שלם?



אינני יודע להגיד כי המצב בקוד מספק לי מענה לכך. כלומר אני סומך על המימוש של שפת Go כי כך זה הדבר, אך אינני בודק זאת בעצמי. כך שאני לא מכסה את כל האפשרויות, ואני יכול להחביא באג פוטנציאלי אשר אינו קשור ישירות לקוד שלי, אך כן יכול להתרחש.

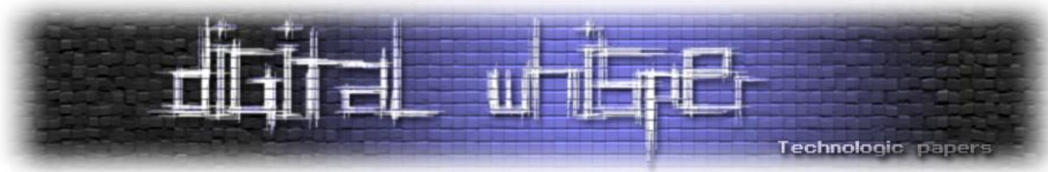
קוד כל כך פשוט כביכול, יכול להחביא בתוכו בעיות אשר ללא ידע או ניסיון קודם יכולים לצוץ ללא הבנה מוקדמת.

התפקיד של הבדיקות הן לא להבטיח כי לא יהיו בעיות, אלא להבטיח כי מה שאני מכיר ויודע שיכול להתרחש לא יתרחש. כך שהבדיקות שומרות על קוד מפני בעיות ידועות ומוכרות, במידה ואינן יוצרות false positive או true negative, אך זה כל התפקיד שלהן.

סיכום

במאמר זה סיפקתי תיאוריה המסבירה מדוע מערך הבדיקות המוכר אינו אמצעי להבטיח כי אין באגים או בעיות בקוד קיים. ההדגמה הקצרה שלי נועדה לספק גישה נגדית לגישה הרווחת כי תפקיד הבדיקות הוא להבטיח כי אין באגים או בעיות אבטחה.

חשוב להדגיש כי אין מאמר זה בא להציע כי מערכי בדיקות הוא דבר מיותר או שניתן להתעלם מהצורך בו, אלא תפקיד המאמר הוא להסביר מה המקום של מערך בדיקות, וכי אין לבלבל בין השימוש במערך בדיקות לבין הרצון להבטיח כי אין באגים או בעיות.



דברי סיכום

בזאת אנחנו סוגרים את הגליון ה-104 של Digital Whisper, אנו מאוד מקווים כי נהנתם מהגליון והכי חשוב- למדתם ממנו. כמו בגליונות הקודמים, גם הפעם הושקעו הרבה מחשבה, יצירתיות, עבודה קשה ושעות שינה אבודות כדי להביא לכם את הגליון.

אנחנו מחפשים כתבים, מאיירים, עורכים ואנשים המעוניינים לעזור ולתרום לגליונות הבאים. אם אתם רוצים לעזור לנו ולהשתתף במגזין Digital Whisper - צרו קשר!

ניתן לשלוח כתבות וכל פניה אחרת דרך עמוד "צור קשר" באתר שלנו, או לשלוח אותן לדואר האלקטרוני שלנו, בכתובת editor@digitalwhisper.co.il.

על מנת לקרוא גליונות נוספים, ליצור עימנו קשר ולהצטרף לקהילה שלנו, אנא בקרו באתר המגזין:

www.DigitalWhisper.co.il

"Talkin' bout a revolution sounds like a whisper"

הגליון הבא ייצא ביום האחרון של חודש מרץ.

אפיק קסטיאל,

ניר אדר,

28.02.2019