מבוא ללמידה חישובית: מטלת מנחה (ממ"ן) 13 - נדב כחלון; דוח שאלה 4

במסמך זה אפרט על הפתרון שלי לשאלה זו - האימפלמנטציה, וניתוח התוצאות.

<u>חלק A</u>

בחלק זה מימשתי את אלגוריתם SMO לאימון Soft-Margin-SVM. הקוד לחלק זה מופיע בקובץ partA.py המצורף, ומכיל את מימוש המחלקה SVM שתפקידה לייצג מודלים כאלו (כמסווגים בינאריים בלבד), ולאמן אותם בעזרת אלגוריתם זה.

כדי להשתמש במחלקה יש ליצור אובייקט SVM בעזרת פונקציית הבנאי. הקלט לפונקציה זו הוא ה-hyper-parameters לפיהם נבנה המודל, וסט האימון אליו הוא יותאם (הפרמטרים KKT_tol ו-signif_eps מוסברים בתיעוד, ואסביר את התפקיד שלהם בהמשך כשאפרט על המימוש). כדי להתאים את המודל לסט האימון, יש לקרוא לפונקציה fit. לבסוף - כדי להעריך את פלט המודל על כל קלט נתון, יש לקרוא לפונקציה evaluate. הממשק שיוצרות הפונקציות במחלקה והמבנה המדויק של הקלט והפלט שלהן מפורטים היטב בקובץ partA.py.

האלגוריתם והמימוש

המימוש של אלגוריתם SMO במחלקה לאימון המודל (ראה פונקציה fit) בנוי ישירות על האלגוריתם והניתוח המעמיק שעשה J. C. Platt ב-1998 [1].

האימון מתחיל בפונקציה fit הפונקציה מכילה לולאה חיצונית, וזוג לולאות פנימיות שרק אחת מהם רצה בכל איטרציה של הלולאה החיצונית. איטרציות הלולאה החיצונית מהוות ריצות מלאות על קבוצת נתוני הקלט ואופטימיזציה של המודל - איטרציות אלו ממשיכות עד שלא ניתן לבצע עוד אופטימיזציה משמעותית (זהו מינוח מעורפל אך הוא יתבהר בהמשך). במצב זה נאמר שהמודל **סיים להתכנס**. איטרציות הלולאות הפנימיות אחראיות על האופטימיזציות ה"אטומיות" הללו - כל איטרציה כזו מנסה "לאפטם" את המודל לפי 2 כופלי לגראנז' בלבד באופן אנליטי, אם אופטימיזציה כזו אפשרית / משמעותית מספיק ואחד מכופלי הלגראנז' מפר את תנאי KKT. בעזרת משפט שהוכיחו Osuna, et al. ב-1997 [2] לפיו כל עוד "נאפטם" את המודל לפי אוסף דוגמאות שאחד מהם לפחות מפר את תנאי KKT, פונקציית המטרה תקטן ותתכנס לנקודה בה כל התנאים מתקיימים - מה שמבטיח את נכונות האלגוריתם.

הסיבה שישנן **שני** לולאות פנימיות נובעת מיוריסטיקה לפיה יותר סביר שדוגמאות אימון **לא-חסומות** (non-bound) - כאלו שכופל לגראנז' המתאים להם אינו 0 או C, יפרו את תנאי KKT, לפיכך האלגוריתם בוחר להתרכז בהן. ככל שהאלגוריתם מתקדם, סביר שדוגמאות חסומות יישארו חסומות, ואילו דוגמאות לא-חסומות יהפכו לחסומות ככל שתהליכי אופטימיזציה קורים. הוא "יאפטם" את המודל לפי דוגמאות אלו קודם (יחזור על הלולאה הפנימית השניה שוב ושוב - כל עוד examine_all=False), עד שכולן יקיימו את תנאי KKT. ברגע שסיימנו לטפל בהם - נעבור לבדוק את סט האימון כולו בלולאה הפנימית הראשונה, ומיד נחזור לתת-הסט של הדוגמאות הלא-חסומות. נסיים רק כאשר כל דוגמאות האימון מקיימות את תנאי KKT או שלא ניתן לבצע אופטימיזציה משמעותית יותר כלל (את זה כמובן נדע לאחר איטרציה של הלולאה הפנימית הראשונה).
בחרתי להשתמש ביורסטיקה הזו מכיוון ש-Platt ב-[1] בחר להשתמש בה, ולדבריו היא מניבה תוצאות טובות.

הטיפול בכל דוגמה שנבחרת באיטרציות הלולאות הפנימיות של fit קורה בפונקציה examine_example. פונקציה זו אחראית לבחון את הדוגמה, לבדוק האם היא מפרה את תנאי KKT (עד כדי tolerance מסוים

שמהווה פרמטר-על למודל - self.KKT_tol), ואם כן - לנסות "לצוות" אותה עם דוגמה נוספת עליהן תתבצע אופטימיזציה "אטומית" כזו - ולקרוא ל-take_step עליה אדון בהמשך.

גם פה אנחנו זקוקים ליורסטיקה לבחירת הדוגמה השניה שהאלגוריתם "יצוות" לדוגמה הראשונה. למעשה ישנו מבנה הירכי של יוריסטיקות לפיהן אנחנו פועלים - והמטרה הכללית היא לבחור דוגמה שתמקסם את השינוי בפונקציות המטרה, ותקרב אותנו מהר יותר להתכנסות (שמובטחת לנו ממשפטו של Osuna ב-[2]).

- היוריסטיקה הראשונה מנסה למקסם את השינוי בכופל לגראנז' השני, כדי למקסם את השינוי בפונקציית המטרה. שינוי זה פרופורציונאלי להפרש בין **השגיאות** של הדוגמה הראשונה לפיה עושים אופטימיזציה לשניה. לפיכך - אנחנו צריכים לבחור את הדוגמה שהשגיאה שלה הכי רחוקה מהשגיאה של הדוגמה הראשונה. חישוב השגיאות כרוך בחישוב פלט המודל - וזהו תהליך יקר. לפיכך אנחנו שומרים **מטמון שגיאות E_cache** עבור השגיאות של המודל על דוגמאות האימון. מיד אסביר לעומק עליו, אבל בינתיים נוכל להשתמש בו לבחור את הדוגמה בעלת השגיאה הרחוקה ביותר מהשגיאה של הדוגמה הראשונה.
- אם היורסטיקה הראשונה לא עבדה (לא יכולנו לבצע אופטימיזציה משמעותית מספיק עם הדוגמה הזו), ננסה במקום את כל הדוגמאות הלא-חסומות (בסדר אקראי - כדי שלא תיווצר הטייה לדוגמאות הראשונות בסט האימון), עד שנקבל הצלחה. ננסה דווקא את הדוגמאות הלא-חסומות מאותה סיבה - סביר יותר שהן יפרו את תנאי KKT, ואולי יהפכו לדוגמאות חסומות לאחר מכן.
- אם גם זה לא עבד - ננסה את כל שאר הדוגמאות בסדר אקראי (מאותה סיבה).

גם בהירכיית היורסטיקות הזו בחרתי מכיוון שלדבריו של Platt במאמרו [1], היא מניבה תוצאות טובות.

מטמון השגיאות: מכיוון שאנחנו צריכים לחשב שגיאות של המודל בהמון מקומות באלגוריתם (בבדיקה האם דוגמה מסוימת מפרה את תנאי KKT ובבחירת הדוגמה השניה לאופטימיזציה) נבחר לשמור בזיכרון את השגיאה של המודל עבור דוגמאות האימון.
השינויים במודל באים לידי ביטוי בשינויים של שני כופלי לגראנז' (מכיוון שכל אופטימיזציה "אטומית" מתייחסת רק לשניים כאלו) - ולכן פונקציית העדכון של המטמון, update_E_cache, בנויה לטיפול בשינויים כאלו. העדכון מתבצע ישירות מהאופן בו הפלט של הSVM מחושב (בצורה ליניארית בכופלי הלגראנז'), והשינוי באותם זוג כופלי לגראנז'.
<u>נקודה מעניינת</u>: למעשה Platt ב-[1] הציע לשמור מטמון שגיאות רק לדוגמאות הלא-חסומות, אך מהניסויים שערכתי (יחד עם הקוד מהחלק השני, שאפרט עליו בהמשך), קיבלתי מהירות ריצה גבוהה יותר כאשר שמרתי את ערכי השגיאות לכל הדוגמאות - להערכתי, מכיוון שגם בדוגמאות החסומות נעשה כנראה שימוש לעתים לא רחוקות.

אחרי שנבחרה הדוגמה השניה לאופטימיזציה, אנחנו קוראים לפונקצייה take_step שאחראית על ביצוע האופטימיזציה. הפונקציה קוראת ל-calc_new_alphas שעושה את החישובים ההכרחיים (עליה אפרט בהמשך). לאחר מכן take_step מעדכנת את הערכים המתאימים במודל, וקוראת ל-update_E_cache לעדכון מטמון השגיאות, כמפורט קודם.

calc_new_alphas מחשבת את הערכים האופטימליים החדשים לכופלי הלגראנז' מעליהם אנחנו עושים אופטימיזציה, יחד עם הערך החדש של ה-threshold. אין הרבה מה לפרט על הפונקציה - מדובר במספר חישובים מתמטיים שעורכים באופן אנליטי את האופטימיזציה לפי זוג הדוגמאות, כפי שפיתח אותה Platt במאמרו [1] (ואנחנו, בשאלה 1 ^_^). גם ערך ה-threshold החדש מחושב כמפורט במאמר הנ"ל - אנחנו בוחרים דוגמה לא-חסומה ומעדכנים אותו לפיה, ולפי תנאי KKT שמבטיחים שתהיה עבורה שגיאה 0. נקודה שחשוב לשים לב אליה היא הבדיקה שהעדכון אכן משמעותי מספיק בשורה 131 של הקובץ (partA.py), שם אנחנו משתמשים בפרמטר-העל signif_eps לבדיקה האם העדכון לא מספיק משמעותי. הבדיקה הזו עוקבת ישירות לבדיקה שציין Platt במאמרו [1], כחלק מהפסאודו-קוד שלו לאלגוריתם.

אציין פה שכל הפרטים הטכניים הקטנטנים של המימוש מתועדים היטב בקוד עצמו. לא אחזור עליהם פה (כי הם מאוד רבים ופחות רלוונטיים), אך הבודק מוזמן לעיין בהם בקוד ^_^

אופטימיזציה למקרה של קרנל לינארי

במהלך האלגוריתם אנחנו צריכים להעריך את פלט המודל על וקטור x מספר פעמים בעזרת הפונקציה evaluate, כדי לחשב את השגיאות של המודל על נתוני האימון. במקרה הכללי, נצטרך לחשב את ערך הקרנל בין x לבין N דוגמאות האימון. לעומת זאת, במקרה הקרנל הליניארי אפשר לעשות זאת תוך חישוב מכפלה פנימית אחת בלבד - בין וקטור משקולות w לקלט x (במקרה הכללי וקטור המשקולות w יהיה ממימד פלט הקרנל - שיכול להיות עצום ואפילו אינסופי מה שהופך את הפתרון הזה ללא מעשי).

זהו שיפור עצום! לפיכך, במקרה הקרנל הליניארי אנחנו שומרים במודל גם וקטור משקולות w, ומשתנה בוליאני is_linear שבעזרתו אנחנו יודעים שהקרנל ליניארי, ואנחנו יכולים להשתמש באופטימיזציה הזו. הוקטור w מתעדכן בכל עדכון של המודל - ראה בפונקציה take_step, שורה 202 בקובץ partA.py. העדכון הוא מידי ומשקף את השינוי ב-w כתלות בשינוי בכופלי לגראנז' המתאימים, בהתאם לקשר הלינארי הישיר בין w לכופלי לגראנז' (המתואר בתנאי KKT).

הליבה של השיפור הזה באה לידי ביטוי בפונקציה evaluate - שבמקרה הקרנל הליניארי מחשבת את פלט המודל בעזרת המכפלה הפנימית של הקלט x ווקטור המשקולות w (פחות ה-threshold, כמובן). במקרה האחר - אנחנו מחשבים את ערך הקרנל בין x לכל אחד מ-N נתוני האימון, בלית ברירה.

יתרה מזאת - בפונקציית עדכון מטמון השגיאות update_E_cache, נוכל לחשב את השגיאות החדשות בעזרת הכפלה ישירה אחת ויחידה בוקטור המשקולות w, לכן עדיף לעדכן את השגיאות ישירות בעזרת evaluate.

כמו תמיד - גם הרעיון הזה הגיע ממאמרו של Platt ב-[1] אותו ציינתי כבר מספר פעמים.

נקודות הנוגעות באתחול המודל

את המודל אנחנו מאתחלים בפונקציית הבנאי __init__ כך שגם ה-threshold וגם כל כופלי לגראנז' הם 0. הבחירה הזו נועדה לעשות לנו חיים קלים ואין לה סיבה מתמטית מתוחכמת - היתרון של הבחירה הזו הוא שכל הדוגמאות הן חסומות בהתחלה (כי כל כופלי לגראנז' הם 0). בהתאם - הפלטים הראשוניים של הSVM כולם הם 0. לפיכך השגיאות הראשוניות הן בדיוק מינוס ערכי המטרה, וככה אנחנו מאתחלים את מטמון השגיאות. בהתאם, כמובן, וקטור המשקולות w מאותחל ל-0 (כי הוא ליניארי בכופלי לגראנז' המאופסים).

נקודה נוספת היא בחירת ערכי ברירת-המחדל ל-signif_eps ו-KKT_tol, שקיבלו שניהם את הערך 0.001. המאמר [1] של Platt הציע לבחור את KKT_tol כ-0.001, ובאופן שרירותי בחרתי גם את signif_eps להיות 0.001. קיבלתי תוצאות לא רעות בכלל (בחלק הבא), לכן הרשיתי לעצמי להשאיר את זה ככה.

ואם כבר ציינו אותו, בואו נעבור לחלק הבא :)

חלק B

חלק זה משתמש בסט הנתונים המפורסם Iris לבחינת המודל שלנו. סט זה הוצג לראשונה במאמרו של הסטטיסטיקאי והביולוג הבריטי רונלד פישר ב-1936 במאמרו [3] - "The use of multiple measurements in taxonomic problems". סט הנתונים מכיל 50 דוגמאות משלושה זנים של פרח האירוס (setosa, virginica, versicolor). ארבעה פיצ'רים נמדדו עבור כל דוגמה: האורך והרוחב של עלי הגביע ועלי הכותרת (בסנטימטרים).

בחלק זה אנחנו נשתמש במסווג multiclass המבוסס על מסווגי SVM שמימשנו בסעיף הקודם כדי לסווג את הדוגמאות בבסיס הנתונים הזה לשלושת הזנים. על תפקיד הסיווג לכמה מחלקות אחראית המחלקה MulticlassSVM, שמשתמשת במסווגים מהמחלקה SVM למימוש מסווג multiclass באסטרטגיית "One-vs-All" (פירוט בהמשך).

הקוד לחלק זה מופיע בקובץ partB.py המצורף.

קצת על המימוש של MulticlassSVM לפני שנתחיל

כדי לסווג קלטים ל-T מחלקות, המסווג שלנו ישתמש ב-T מסווגי SVM בינאריים (מהסעיף הקודם) שנשמרים במערך bin_SVMs (ראה פונקציית הבנאי __init__ של המחלקה). המסווג הבינארי ה-t נבנה ומותאם כדי לסווג בין 2 המחלקות הבאות: נתונים במחלקה t (עבורם y=+1), ונתונים שאינם במחלקה t (עבורם y=-1). גם את זה ניתן לראות בפונקציית הבנאי (בשורה 57 שיוצרת את וקטור המטרה עבור המסווג הבינארי ה-class_num). מעבר לזה - המחלקה SVM מהסעיף הקודם תטפל בכל תהליך האימון. ואכן - פונקציית האימון fit פשוטה למדיי.

לגביי הסיווג עצמו: המסווג הבינארי ה-t ב-bin_SVMs הותאם כדי להפיק ערכים גדולים יותר (חיוביים) לקלטים במחלקה ה-t, וערכים קטנים יותר (שליליים) לקלטים שאינם במחלקה ה-t. לפיכך - הגיוני לצפות שאם קלט שייך למחלקה t אז המסווג הבינארי של המחלקה ה-t יפיק עבורו פלט גבוה, ושאר המסווגים הבינאריים יפיקו עבורו פלט נמוך. מכאן נובעת האסטרטגיה שלנו לסיווג קלט בפונקציה classify: אנחנו בוחרים לסווג קלט מסוים למחלקה עבורה המסווג הבינארי הפיק את הקלט הגבוה ביותר. המסווג הזה "הכי האמין" בקלט - הוא שלח אותו הכי רחוק לצד החיובי של המרחב (או לפחות הכי קרוב לצד החיובי). לפיכך, גם המימוש של classify מאוד פשוט (ומשתמש ב-evaluate של המחלקה SVM).

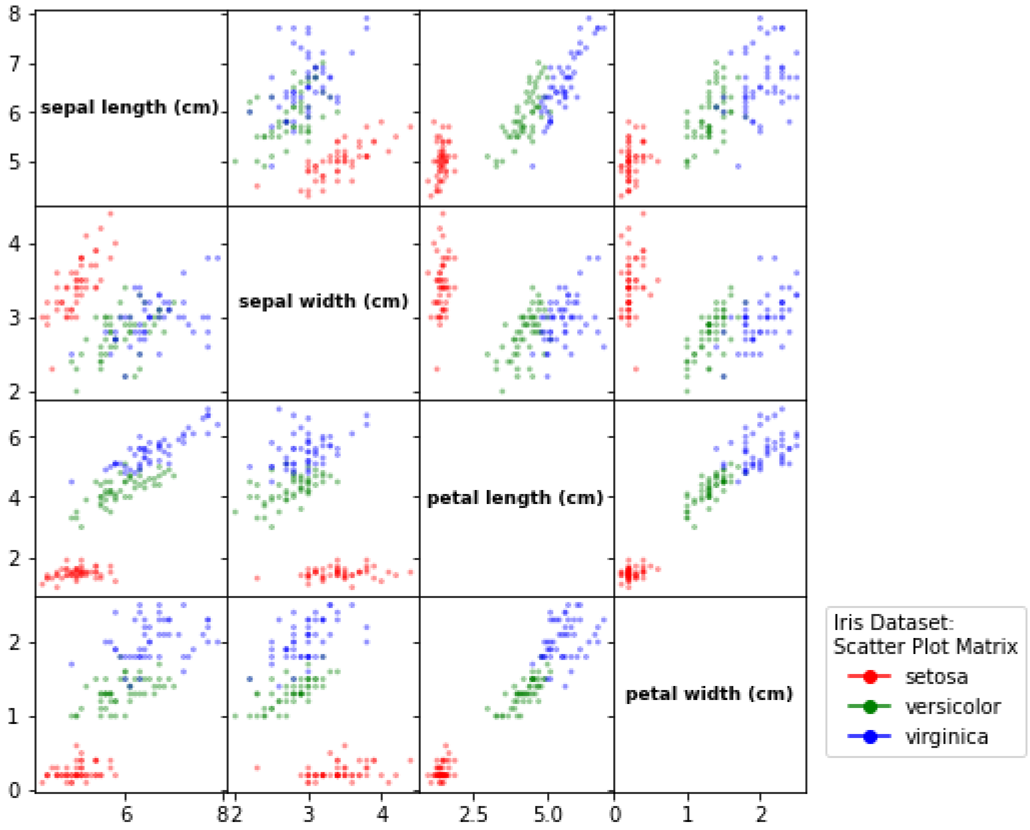
כמו כן, המימוש של calc_accuracy - הפונקציה שנועדה לחשב את הדיוק של המסווג על קבוצות ולידציה, הוא מיידי: אנחנו פשוט משתמשים ב-classify לסווג את הדוגמאות ומחזירים את אחוז הדוגמאות שסווגו לא נכון.

לגביי הפונקציה האחרונה במחלקה - calc_confusion_mat - עליה אדון בסעיף הרלוונטי בהמשך (סעיף c).

סעיף a - הצגת בסיס הנתונים Iris

בסעיף זה נציג את בסיס הנתונים Iris עליו אנחנו נעבוד בחלק הזה של השאלה בעזרת **מטריצת דיאגרמות פיזור**. לחלק זה אחראית הפונקציה scatter_plot_mat בקובץ partB.py, ושורות 256-261 באותו הקובץ (בתוכנית הראשית). אין הרבה מה לפרט - מדובר בסך הכל בהמון טיפולים קוסמטיים על התרשים שלנו, פחות רלוונטי לחקר עצמו.

מטריצת דיאגרמות הפיזור שקיבלנו:



ניתן לראות בבירור שזן ה-setosa (באדום) הוא הכי ניתן להפרדה מבין הזנים האחרים: הצטברות הנקודות האדומות ניתנת להפרדה לינארית בכל דיאגרמת פיזור במטריצה (לפעמים אפילו די בקלות - כמו ביחס בין הרוחב והאורך של עלי הכותרת, מה שמעיד על כך שעלי הכותרת של הזן הזה נבדלים היטב). לעומתו, הנתונים על הזנים versicolor (בירוק) ו-virginica (בכחול) קרובים מאוד ואף מעורבבים אחד בשני. אף על פי כן ניתן לראות הפרדה מסוימת גם בין versicolor ל-virginica בדיאגרמות הפיזור, אם כי חלשה הרבה יותר (בעיקר בדיאגרמה המייצגת את הקשר בין הרוחב לאורך עלי הגביע של הפרחים)

סעיף b - אימון MulticlassSVM עם קרנל לינארי, תוך שימוש בולידציה לבחירת הפרמטר C

לחלק זה אחראיות שורות 263-290 בקובץ partB.py (בתוכנית הראשית).
ראשית אנחנו מפצלים את הנתונים לאימון, בחינה, ו-ולידציה בהתאם לקבועי החלוקה שהגדרתי לפני תחילת התוכנית הראשית (train_part, val_part, test_part - שורות 243-246).
לאחר מכן אנחנו מאמנים מודלים של MulticlassSVM עם קרנל ליניארי כמה פעמים על ערכי C שונים, ובוחנים את הביצועים שלהם על סט הוולידציה (בעזרת הפונקציה calc_accuracy של MulticlassSVM).
ערכי C השונים בהם אנחנו משתמשים מוגדרים גם הם לפני התוכנית הראשית, במערך C_lisr (שורה 248).
לבסוף - אנחנו בוחרים את ערך ה-C שהביא לנו את אחוזי הדיוק הטובים ביותר ומאמנים עליו מודל אחרון - linear_model.

<u>כמה מילים על בחירת ערכי C</u>: בחרתי בחזקות 10 כדי לחקור טווח יחסית גדול של ערכים (מסדרי גודל **שונים**) ולנסות לראות שינוי משמעותי מערך לערך. ניסוי ותהייה הוביל אותי להסיק שזהו טווח סביר (לא נמוך מדי ולא גבוה מדי), שנותן כמה תוצאות מגוונות.

<u>כמה מילים על שימוש באחוז דיוק להערכת המודלים</u>: אמנם אחוז הדיוק יכול להיות מדד מטעה להערכת מסווגי multiclass, אבל הרשיתי לעצמי להשתמש בו היות ובסיס הנתונים מכיל מספר שווה של דוגמאות לכל מחלקה (50), ככה שאין את החשש שמחלקה מסוימת תקבל משקל גדול יותר משמעותית ממחלקה אחרת.

<u>כמה מילים על חלוקת סט הנתונים</u>: בחרתי בחלוקה של 50% לאימון. 30% לבחינה, ו-20% לוולידציה - בשונה מ"כלל האצבע" שהציעו בתרגיל. הסיבה לכך היא שכאשר ניסיתי לעבוד עם 20% בחינה (ו-60% אימון), התוצאות היו מושלמות היות ובסיס הנתונים מאוד מאוד קטן (קיבלתי 100% דיוק ו-100% רגישות לכל מחלקה) - לא היה עניין בביצועים הללו! לכן הגדלתי את סט הבחינה, כדי לאפשר למודל להתנסות בדוגמאות מגוונות מספיק. (אגב - ניסיתי לאזן את זה ולאפשר 30% לוולידציה ורק 40% לאימון, אך לא הבחנתי בשינוי ולכן החזרתי את זה ל-50% אימון ו-20% וולידציה).

תוצאות:

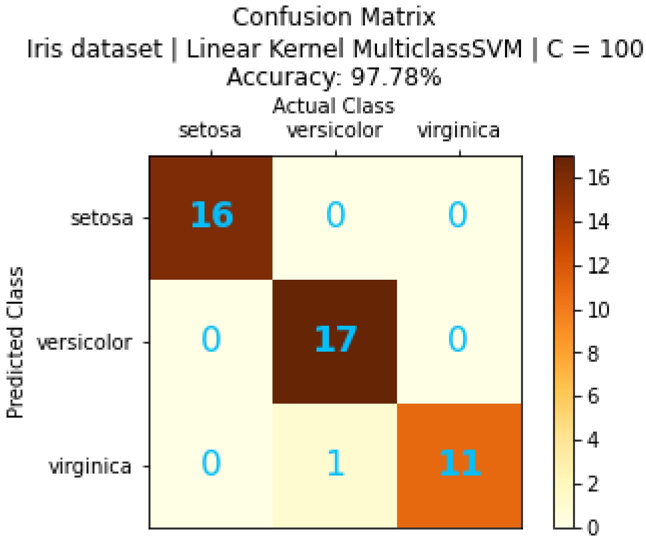
C	0.1	1	10	100
Accuracy (on validation set)	80.65%	96.77%	93.55%	100%

קיבלנו את התוצאות הגבוהות ביותר עבור C=100. לעומת זאת - עבור C קטן (0.1) קיבלנו תוצאות פחות טובות בפער ענק (כ-20% פחות מהמקרה הטוב ביותר, וכ-13% פחות מהמקרה הגרוע ביותר הבא).
להערכתי, קיבלנו את התוצאה הזו מכיוון שסט הנתונים גם ככה לא ניתן להפרדה ליניארית - ואם בקושי "נקנוס" שגיאות בסיווג (כלומר - נבחר C קטן), נקבל underfitting משמעותי.

סעיף c - דיוק המודל ו-confusion-matrix

לסעיף זה אחראיות הפונקציות calc_confusion_mat במחלקה MulticlassSVM,
ו-plot_confusion_mat.
הפונקציה calc_confusion_mat מחשבת את ה-confusion-matrix של מסווג multiclass על סט בחינה מסויים. החישוב הוא פשוט מאוד - מחשבים את הסיווג של כל דוגמת בחינה, וסופרים את הזוגות השונים של סיווג משוער וסיווג אמיתי. ערכים אלו נכנסים לכניסות המתאימות במטריצה.
הפונקציה plot_confusion_mat אחראית על שרטוט המטריצה בדיאגרמה, יחד עם חישוב הדיוק של המודל. ברובה היא מכילה פרטים קוסמטיים שלא אפרט עליהם. החלק היחיד שרלוונטי לנו הוא חישוב הדיוק מתוך המטריצה (שורה 142): הדיוק הוא מספר הסיווגים הנכונים (סכום איברי האלכסון הראשי של המטריצה), חלקי מספר הסיווגים בסה"כ (סכום כל הכניסות במטריצה).

התוצאה (בעמוד הבא):

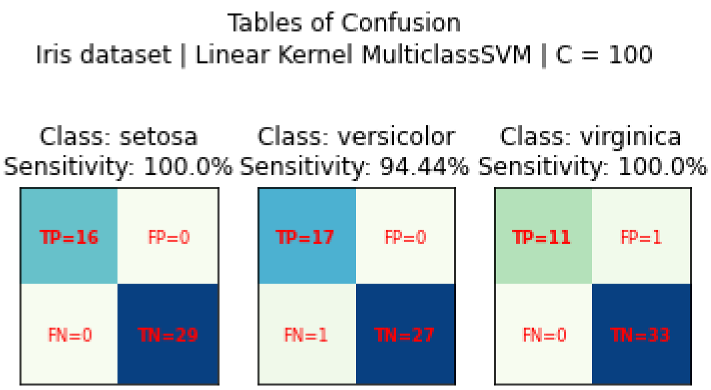


ניתן לראות שביצועי המודל היו כמעט מושלמים, אבל הוא טעה בסיווג אחד: אחד מנתוני הקלט סווג כשייך לזן virginica, כשלמעשה הוא שייך לזן versicolor. עובדה זו לא אמורה להפתיע אותנו - אחרי הכל ראינו בסעיף a שלא פשוט להפריד בין המחלקות האלו (לעומת ההפרדה הברורה בין setosa למחלקות האחרות). עם זאת, ההפרדה לא הייתה נוראית - ברוב הזמן היא הצליחה. גם זה לא כל כך מפתיע - כמו שראינו בסעיף a, אפשר (במידה מסוימת) להבחין בהפרדה בין מחלקות versicolor ו-virginica במטריצת דיאגרמות הפיזור. הדיוק שקיבלנו היה קצת נמוך מהדיוק לקבוצת הוולידציה (שהיה 100%). זו לא תוצאה מפתיעה כמובן. בסופו של דבר הרי בחרנו את המודל שהפיק את התוצאה הגבוהה ביותר לקבוצת הוולידציה - והגיוני לחשוב שלמודל יש bias לטובתה. עם זאת - הדיוק עדיין מאוד גבוה.

סעיף d - רגישות ו-table-of-confusion

לסעיף זה אחראיות הפונקציות plot_confusion_table ו-plot_confusion_tables.
הפונקציה plot_confusion_table אחראית על שרטוט table-of-confusion יחיד. חישוב הכניסות בטבלה מתבצע בעזרת ה-confusion-matrix: אנחנו יכולים לראות כמה דוגמאות סווגו למחלקה שלנו או מחוץ למחלקה שלנו, כתלות ב-האם הן באמת שייכות למחלקה שלנו או לא. מעבר לזה הפונקציה כוללת פרטים קוסמטיים בלבד, שלא אפרט עליהם. כמו כן, הפונקציה plot_confusion_tables בסך הכל מטפלת בשרטוט מספר tables-of-confusion ביחד, לצורך נוחות התוכנית הראשית.

תוצאות:



כמו שסביר לצפות, קיבלנו טבלה מושלמת עבור setosa - המחלקה שכפי שראינו בסעיף a ניתנת להפרדה בצורה הטובה ביותר מהמחלקות האחרות. לעומתה, הטבלאות של versicolor ו-virginica לא מושלמות - יש בהם בלבול אחד (false negative ב-versicolor ו-false positive ב-virginica) שכמובן מתאים לבלבול

האחד שראינו ב-confusion matrix בסעיף הקודם בין המחלקות (בו דוגמה השייכת ל-versicolor סווגה כשייכת ל-virginica). זה כמובן מכיוון שלא פשוט להפריד בין המחלקות הללו, כפי שראינו במטריצת דיאגרמות הפיזור בסעיף a. אף על פי כן, ראינו שכן אפשר להבחין (במידה מסוימת) בהפרדה בין המחלקות - ולכן הטבלאות שלהן לא נוראיות, וקיבלנו רגישות יחסית גבוהה גם ב-versicolor.

סעיף e - הכל מחדש, רק ל-RBF

בסעיף זה נערוך את כל הניתוחים מהסעיפים הקודמים, רק עם מסווג שפועל על קרנל RBF. לחלק זה אחראיות שורות 300-332 בקובץ partB.py. לא אחזור על איך שכל החישובים פועלים ומתבצעים - הכל קורה בדיוק כמו בסעיפים הקודמים, רק עם יצירת מודל MulticlassSVM עם קרנל RBF, וערך גאמא מתאים.

את הולידיציה אנחנו עושים על זוגות של פרמטר C ופרמטר גאמא (של הקרנל), ובוחרים את הזוג שהניב את הדיוק הגבוה ביותר. את C אנחנו בוחרים מבין הערכים השונים ב-C_list (שורה 248), ואת גאמא אנחנו בוחרים מבין הערכים השונים ב-gamma_list (שורה 249). פירטתי בסעיף b על כל הבחירות הקודמות שעשיתי (הבחירות לטווח הערכים של הפרמטר C, שימוש בפונקציית דיוק להערכת המודלים, והחלוקה של סט הנתונים), ונשאר לפרט על הבחירה לטווח הערכים של הפרמטר גאמא.

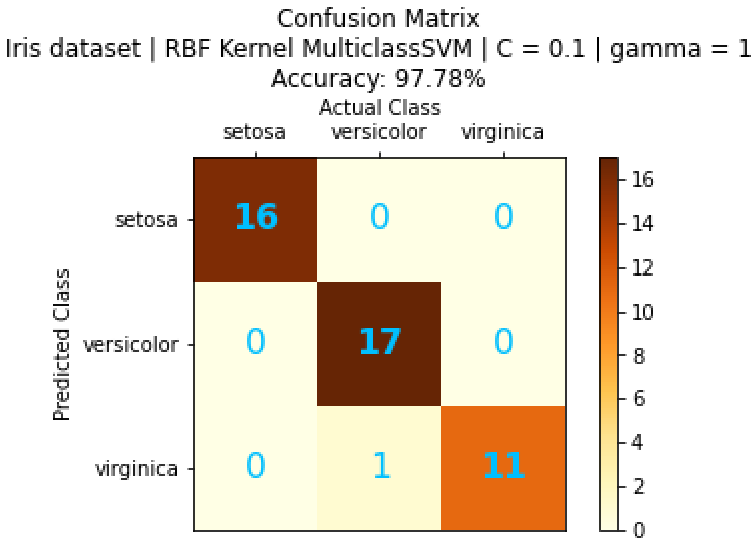
גם פה, בחרתי בגידול אקספוננציאלי בין הערכים השונים כדי לחקור טווח גדול וסדרי גודל שונים, ולראות ביניהם שינוי משמעותי. מניסוי ותהיה, שמתי לב שהתוצאות מאוד רגישות לשינויים בפרמטר הזה, לכן החלטתי על חזקות של 2 (בסיס יותר קטן מבסיס 10 שהשתמשתי בו בבחירת ערכי C).

תוצאות הדיוק של המודלים השונים על סט הוולידציה:

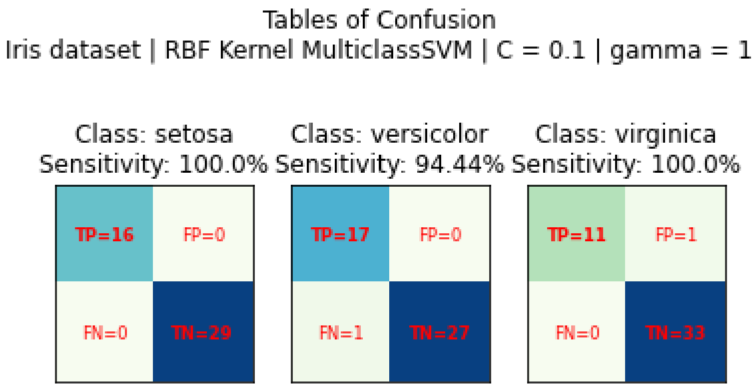
gamma \ C	**0.1**	**1**	**10**	**100**
0.25	93.55%	100%	100%	96.77%
0.5	93.55%	100%	100%	96.77%
1	100%	100%	96.77%	96.77%
2	100%	100%	96.77%	96.77%

נראה שככל שערך C גדול יותר, הקרנל עובד טוב יותר עם ערכי גאמא קטנים יותר. בנוסף, הוא עובד הכי טוב עם ערכי C שאינם גדולים מדי ואינם קטנים מדי (לא 100 או 0.1). עבור C=1 קיבלנו תוצאות מושלמות לכל ערך של גאמא, לעומת זאת האלגוריתם בוחר את התוצאה הטובה ביותר הראשונה שהוא מצא - לכן להמשך הדרך ייבחר הזוג C=0.1 ו-gamma=1.

תוצאות ה-confusion matrix (בעמוד הבא):



תוצאות ה-tables of confusion:



קיבלנו טבלאות זהות - כנראה מאותן הסיבות עליהן פירטתי בסעיפים c ו-d.

השוואה בין תוצאות הקרנל הליניארי לתוצאות הקרנל RBF:

לצערנו, מכיוון שבסיס הנתונים קטן מאוד לא קיבלנו **בכלל** הבדל בין טבלאות ה-confusion matrix ו-tables of confusion, וקשה מאוד להסיק מהן איזושהי מסקנה מעניינת שמבחינה בין הקרנלים. אולי כן נוכל להבחין בעובדה שהמעבר לקרנל rbf, שהיה אמור להיות פחות רגולרי ולאפשר מודל יותר מורכב מסתם מפרידים ליניאריים (שראינו שבלתי אפשרי ליצור כדי להבחין בין מחלקות virginica ל-versicolor), לא הוביל לשיפור כלל. מכאן אפשר אולי להסיק שאין יתרון משמעותי במדד הדיוק ל-rbf לעומת קרנל ליניארי על סט הנתונים הזה, ובכל מקרה עדיף לנו לבחור בקרנל הליניארי שרץ מהר יותר.

אף על פי כן, אם נתבונן בטבלאות המייצגות את אחוזי הדיוק על סט הוולידציה (ראה לעיל), נראה שדווקא כן יש יתרון ברור לקרנל rbf. הקרנל הליניארי ירד עד 80.65% דיוק (עבור C=0.1), בעוד שקרנל RBF לא ירד מ-93.55% ובחצי מהמקרים הגיע לדיוק של 100% על סט הוולידציה (גם עבור C=0.1). ואילו, יתרון זה לא משמעותי בכלל במבחן התוצאה הסופי מכיוון שאנחנו בחרנו את המודלים עם הדיוק הגבוה ביותר על סט הוולידציה - שבשני המקרים היה 100%! אפשר אולי רק להסיק מזה שקרנל RBF, שמאפשר מודלים פחות רגולריים ויותר גמישים מקרנל ליניארי, מתאים יותר **באופן כללי** לסט הנתונים שלנו ללא תלות בפרמטרים שלו. אולי בגלל שראינו שבלתי אפשרי להפריד ליניארית את מחלקות virginica ו-versicolor, זה גורם לכך שלקרנל הליניארי יש חיסרון במקרים מסוימים על הסט הזה (המקרים עם ערך C נמוך שמאפשר יותר שגיאות בסיווג ולא מנסה).

רשימה ביבליוגרפית

[1] Platt, J. C. Sequential minimal optimization: A fast algorithm for training support vector machines. Technical Report MST-TR-98-14. Microsoft Research, (1998).

[2] Osuna, E., Freund, R., Girosi, F. Improved Training Algorithm for Support Vector Machines. Proc. IEEE NNSP ’97, (1997).

[3] Fisher, R. A. The use of multiple measurements in taxonomic problems. Annals of eugenics, 7(2), 179-188, (1936).

בהצלחה ^_^