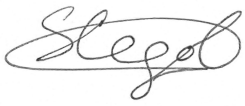
ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

Отчет о программном проекте

на тему: Программа обнаружение изменений на спутниковых данных

Выполнил:



Студент группы БПМИ204 _____ С.А.Седов
Подпись И.О.Фамилия

19.05.2022
Дата

Принял:
Руководитель проекта Родригес Залепинос Рамон Антонио
Имя, Отчество, Фамилия

доцент ФКН, департамент Программной Инженерии
Должность, ученое звание

ФКН НИУ ВШЭ
Место работы (Компания или подразделение НИУ ВШЭ)

Дата проверки 19.05 2022 _____ _____
Оценка (по 10-ти бальной шкале) Подпись

Москва 2022

Содержание

1 Введение

Основной задача, стоявшей передо мной после выбора проекта, была имплементация обнаружения изменений на спутниковых данных с помощью одного из алгоритмов машинного обучения. Начал я с чтения нескольких статей / просмотра видео-уроков [11] [2], описывающих проблематику поставленной задачи, а также современные методы, применимые для её решения.

Для начала давайте разберёмся, зачем вообще нужно классифицировать данные (выделяя изменённые), и почему мы не останавливаемся, просто показав изменённые пиксели на каждой фотографии. Ответ на этот вопрос можно получить, задав себе другой: а когда и как используются данные об изменении местности? Конечно, сложно перечислить все их применения, но я постараюсь назвать хотя бы самые популярные, пришедшие в голову:

- Экологи оценивают вред окружающей среде смотря на изменения лесного /ледяного покрова.
- Определение лесных пожаров, для заблаговременного предотвращения или же симуляции их развития.
- Оценка изменения уровня воды в водоёмах: осушение озёр, выход рек из берегов, скоро, по всей видимости, дело дойдёт и до мирового океана...
- Определение возникающих ураганов, предсказание их маршрута для эвакуации населения
- Наблюдение за ростом городов и симуляция дальнейшего развития - помогает правильным образом развивать инфраструктуру

Посмотрев на все эти примеры, легко подметить - каждый раз после обнаружения изменений, классифицируя пиксели на изменённые и нет, мы хотели бы классифицировать и сами изменения: узнать, какой тип местности изменился и тому подобное. Нам нужна информативность, отсутствующая в обычном сравнении картинок. К тому же на каждом снимке будут собственные условия среды - какие-то помехи, разная контрастность, тени, времена года, etc. Всё это мы хотели бы отбрасывать, постепенно получая всё более точные результаты. Как уже было сказано, за этим чаще всего следует классификация самих изменений, но это уже несколько другая история... А я перехожу к описанию своих действий.

2 Основная часть

Я решил использовать метод Опорных Векторов, более известный как SVM - Support Vector Machine [9]. Во второй части этого отчёта я подробнее расскажу про своё погружение в изучение данного метода. А пока первоначальная задача: перед нами есть классифицированный датасет изображений [1], это обычные rgb-изображения, другими словами при переводе изображения из PIL.Image в numpy.array для каждого пикселя мы получим массив из трёх элементов. Мы хотим взять часть этого датасета для тренировки классификатора, и часть поменьше - для проверки точности. Пускай для тренировки выбрали $count\_images$ изображений размера $m \cdot n$. У нас будет общий массив X_train размером $count\_images \cdot m \cdot n$, причем $X\_train[i] = [\delta_r, \delta_g, \delta_b]$ - попиксельная разница между изображениями. Каждый вектор $X\_train[i]$ назовём sample'ом, каждый элемент этого вектора - feature. В соответствие каждому sample'у ($X\_train[i]$) ставится label ($y\_train[i]$) $= 1$ (пиксель изменён) или -1 (существенных изменений не было). В нашем случае каждое изображение размером $512 \cdot 512 = 262144$ пикселей, что на практике означает чрезмерно большое время обучения модели уже для нескольких десятков изображений.

В обучении классификатора мы бы хотели в первую очередь использовать разные изображения, чтобы меньше зависеть от конкретного типа изменения (в городах появляются дома, в лесах - проплешены и т.п.). Раз мы не хотим критически понижать количество изображей от которого зависит количество сэмплов, то мы вынуждены пожертвовать количеством фич (разные цвета несут меньше информациин нежели разные типы местности). Мы жертвуем трёхцветностью, просто получая изображения в чб или же считая $\sqrt{\delta_r^2 + \delta_g^2 + \delta_b^2}$. Большой разницы между этими двумя подходами нет. Тем не менее, даже уменьшив количество фичей, вычисления всё равно занимают слишком много времени. Я решил бороться с этим дальше - путём кластеризации изображения. Действительно, нам не слишком важны изменения малых группок пикселей, они даже вероятно будут вызывать лишние помехи. Поэтому я разбиваю каждую матрицу изменений на кластеры - делю массив размера $512 \cdot 512$ на кластеры размера $8 \cdot 8$. Посчитав средние изменения по каждому из кластеров, мы получаем новый массив $X\_train\_clusters$ и $y\_train\_clusters$ размера $64 \cdot 64$. Для таких размеров SVM работает уже существенно быстрее.

Итого: мы применили 2 техники, позволивших нам координально сократить время работы программы, лишь немного пожертвовав точностью, а значит и конечной информативностью. Первая - переход от большего числа характеристик к меньшему, отбрасывая малозначительные (в нашем случае это просто переход к чб, но на иных мультиспектральных изображениях это может быть и апроксимация других измерений чем-то общим). Вторая - кластеризация изображений. Однако можно пойти дальше - определить интервал неопределённости. Таким образом, SVM будет обучаться разделению точек на 3 класса: пиксель не изменён / изменён / непонятно, изменён или нет. Есть различные методики определения этого интервала; пожалуй, самая доступная из них - отбрасывание нижних p процентов изменённых точек, где p подбирается вручную (например засчёт кросс-валидации на сетке из различных значений p) и является средним для всех изображений. Усложнённая версия - алгоритм Expectation-Maximization [4], подбирающий p с помощью оценок максимального правдоподобия. В данном проекте я не буду использовать EM, так как это специфика, но в целом упоминуть про него стоило.

Прежде чем перейти к подробному разбору теории по классификации с использованием SVM, я опишу используемые библиотеки, алгоритмы и методы. В моём проекте 2 основных ipynb файла, в первом я имплементировал SVM-classifier самостоятельно и сравнивал его с svm.SVC [8] из sklearn; во втором я занимался непосредственно классификацией изменений на спутниковых данных. Вот общий список используемых мною библиотек:

```
import numpy as np
import pandas as pd # reading data from csv
from PIL import Image # image opening

from sklearn import svm # comparing to sklearn SVM

from sklearn.model_selection import train_test_split # splitting dataframes
from sklearn.model_selection import ShuffleSplit
from sklearn.metrics import accuracy_score, recall_score, precision_score # result estimation
from sklearn.preprocessing import MinMaxScaler # normalization

from scipy.stats import pearsonr # correlation and p-value
import statsmodels.api as sm # finding the p-value

import matplotlib.pyplot as plt # plotting
import math
import os

from scipy.optimize import Bounds, BFGS
from scipy.optimize import LinearConstraint, minimize
✓ 2.7s
```

2.1 Первая часть

Я реализовывал свою вариацию SVM, после чего тестировал её на небольшом датасете, который необходимо было классифицировать. Далее я опишу основные действия по запуску алгоритма классификации.

1. Читаем файл с помощью pandas.read_csv, мапаем labels в $\{-1, 1\}$. Разбиваем данные на X, y.
2. Отбрасываем коррелированные характеристики (features) в X с помощью анализа матрицы ковариаций.
3. Отбрасываем незначительные характеристики в X с помощью нахождения p_value между каждой парой столбцов. Оба пункта мы делаем с помощью scipy.stats.pearsonr

4. Запускаем в кросс-валидацию. Я написал свою, больше для практики. В ней разным образом разбиваем данные на тренировочные и тестовые, считаем average_score из score для всех запускаов (в моём классе с помощью sklearn.metrics.accuracy_score, в sklearn.svm.SVC есть встроенный метод score),

5. Сравниваем полученные результаты, а также время работы алгоритмов

Теперь действия самого алгоритма классификации, в случае использования stochastic gradient descent [10]. От обычного алгоритма спуска последовательно по каждой из координат он отличается тем, что мы выбираем несколько направлений, оптимизируя их одновременно. Причину, по которой покоординатный спуск не работает я опишу несколько позже, в секции разбора алгоритмов.

1. Выставляем изначальные параметры: максимальное количество итераций и cost_threshold_multiplier - пороговое значение относительной разницы cost на данном шаге с предыдущим, с которого мы прерываем градиентный спуск. Также изначально задаём регуляризатор скорости обучения.

2. На каждой итерации рандомным образом выбираем подмножество X_batch и y_batch, пробегая по перестановкам индексов.

3. Считаем gradient_cost на данном шаге по формулам, которую я здесь выписывать не буду, но это подсчёт смещений по частным производным, в данном случае для спуска по нескольким направлениям.

4. Останавливаемся в случае достижения относительного порогового значения.

Также я реализовывал вторую вариацию алгоритма - через сведение к соответствующей задаче оптимизации, после чего использовал scipy.optimize.minimize.

2.2 Вторая часть

Реализация классификации изменений местности с использованием SVM'а. Специфику задачи я описывал выше, здесь же я укажу алгоритм запуска классификатора.

1. Загружаем данные в классе Data. На выбор у нас 2 разных способа загрузки: чёрно-белые изображения или цветные. Перебираем изображения из указанных директорий на указанном интервале. Необходимо нормализовать данные и привести label'ы к нашему формату. Помимо обычного функционала numpy используем os.listdir, PIL.Image.open, sklearn.preproccessing.MinMaxScaler.

2. В случае с цветными данными мы сохраняем попиксельные разности в X, а также правильные label'ы в y.

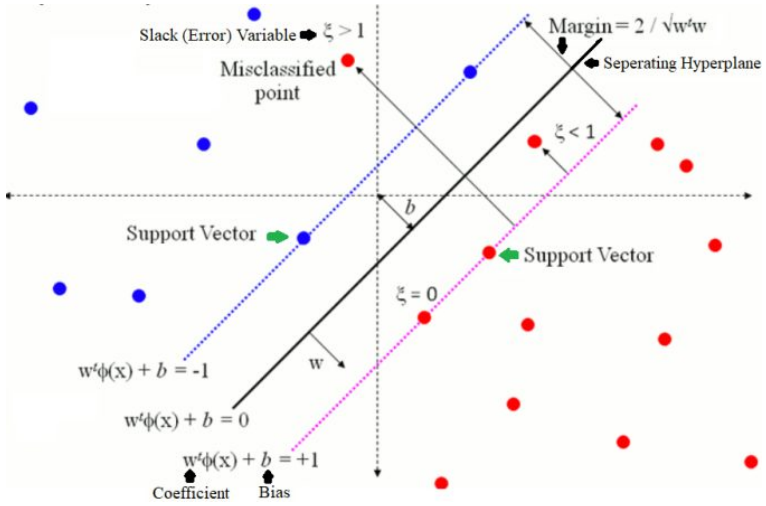
3. Если мы хотим работать с 1 характеристикой, то мы можем либо загрузить чёрно-белое изображение, либо посчитать общую норму вектора из трёх измерений.

4. Будем кластеризовывать данные, в зависимости от указанных размеров. Можно подсчитывать просто средние значения на кластерах, можно использовать EM.

5. Начинаем перебирать параметры, в том числе перебирая количество кластеров. Что мы можем менять: количество изображений, размерность изображений, количество кластеров, тип ядра в svm.SVC, параметры конкретного ядра, параметр trade-off'а. Всё это приводит нас к куче времени ожидания, но в итоге я смог подобрать оптимальные в моём случае коэффициенты.

2.3 Support Vector Machine

Рассмотрим классический пример применения алгоритма. У нас есть датасет из n sample'ов: $\{x_i, y_i\}$, где x_i - вектор features, а $y_i \in \{-1, 1\}$ - классификации векторов. Мы хотим разделить этот датасет гиперплоскостью так, вектора из разных классов были по разные стороны от неё. Причём мы хотим чтобы эта гиперплоскость была как можно дальше от крайних векторов (которые собственно и называются support vectors)/ Для простоты рассмотрим 2-d случай: нужно провести прямую, которая разделит 2 группы векторов "лучшим образом то есть максимизирует расстояние от опорных векторов.



Гиперплоскость задаётся уравнением $w^T \cdot x + b = 0$. Если датасет изначально нормализован, то уравнения для опорных векторов будут такие: $w^T \cdot x + b = \pm 1$, а ширина полосы между опорными векторами разных классов $\frac{2}{||w||}$. Мы хотим максимально расширить эту полосу, имея все точки по 2 стороны от неё. Посмотрев на ширину полосы, становится очевидно, что нужно минимизировать $||w||$. Мы будем минимизировать её квадрат, просто потому что в случае с второй нормой это проще. Таким образом решаем задачу оптимизации $||w||^2 \to min$, имея систему ограничений:

$$\begin{cases} w^T \cdot x_i + b \geqslant 1, \ y_i = 1 \\ w^T \cdot x_i + b \leqslant -1, \ y_i = -1 \end{cases}$$

Теперь представим, что в нашем датасете есть несколько выбросов - векторов, не лежащих рядом со всей своей группой, ведь идеальные результаты бывают редко. Итак, мы хотели бы не учитывать такие выбросы, пренебрегать малой частью датасета. Выходом из данной ситуации будет введение функции потерь - штрафа за нахождение точки внутри нашей полосы.
В случае с SVM-ом лучше всего работает hinge-loss function [5]: $l(x) = max(0, 1 - t \cdot x)$. В нашем случае это будет $max(0, 1 - y_i \cdot (w^T \cdot x_i + b))$. Для наглядности предположим, что $y_i = 1$. Тогда функция будет линейно расти при $w^T \cdot x_i + b \in [0, 1]$. То есть для каждой точки внутри полосы вводится линейный штраф по её удалённости от границ полосы.
Мы имеем право выбирать, насколько сильно стоит штрафовать за нахождение в полосе - в зависимости от практической задачи этот параметр может меняться; если в датасете потенциально есть существенное количество неточных измерений, то штрафовать за неточности классификации нужно слабее, иначе мы выберем за опорные вектора крайние точки - неточные данные. Этот параметр обозначим за λ - коэффициент перед $||w||^2$. Наша итоговая задача выглядит вот так:

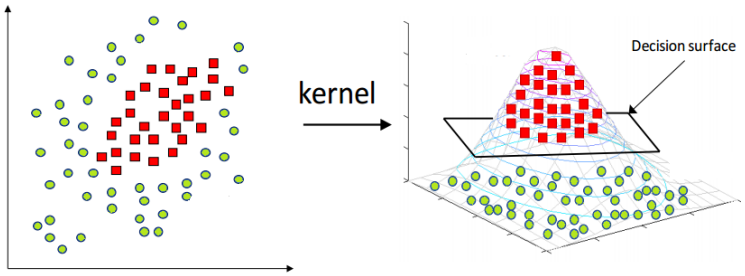
$$minimize \ \lambda \cdot ||w||^2 + \sum_{i=1}^{n} max(0, 1 - y_i \cdot (w^T \cdot x_i + b))$$

Перепишем нашу задачу, введя $\xi_i = max(0, 1 - y_i \cdot (w^T \cdot x_i + b))$

$$minimize \ \lambda \cdot ||w||^2 + \sum_i \xi_i$$

$$with \ constraints: \ \xi_i \geqslant 0, \ y_i \cdot (w^T \cdot x_i + b) \geqslant 1 - \xi_i \forall i \in \{1, ..., n\}$$

И тут стоит подумать о перспективах применения алгоритма. Мы хотели бы разделять датасеты не только гиперплоскостями, но и какими-то кривыми. Как этого можно достичь? Своего рода гениальный ответ на этот вопрос принято называть kernel-trick [6]. Мы отобразим наше пространство в другое большей размерности, посчитав kernel-функции для всех пар векторов. В большем пространстве мы проведём гиперплоскость, которая будет точнее разделять наши точки. При проекции в первоначальное пространство от гиперплоскости останется её сечение с меньшим пространством, то есть какая-то кривая, которая и будет разделять наши точки. Боюсь, без визуализации это очень трудно понять, так что вот она:



Но чтобы применить kernel-trick, мы должны свести задачу к другому виду, в котором зависимости между x_i и x_j будут явно выражены - мы заменим их на $K(x_i, x_j)$, построив желанное отображение. Для этого создателями алгоритма использовался метод множителей Лагранжа и сведение задачи к двойственной задаче минимизации. Это был большой пласт новой теории для меня, но я постарался осознать метод до конца.

Для начала расскажу про функции Лагранжа. Пусть перед нами стоит задача минимизации $f(x)$ с учётом того, что в точках минимума $g(x) \leqslant 0$. Функцией Лагранжа назовём $L(x, \lambda) = f(x) + \lambda \cdot g(x)$, $lambda \geqslant 0$. Тогда минимум для $f(x)$ с учётом ограничения на $g(x)$ будет достигаться в стационарной точке функции Лагранжа. Давайте получим некоторую интуицию, почему это так. Для простоты я рассматриваю лишь одномерный случай с одним ограничением. В многомерных вариантах рассуждения полностью аналогичны. Пускай мы находимся в стационарной точке:

$$\begin{cases} \frac{\partial L(x,\lambda)}{\partial \lambda} = g(x) = 0 \\ \frac{\partial L(x,\lambda)}{\partial x} = f'(x) + \lambda \cdot g'(x) = 0 \ \rightarrow \ f'(x) = -\lambda \cdot g'(x) \end{cases}$$

Раз $g(x_0) = 0$, значит в локальном минимуме по x для $L(x_0, \lambda_0) = f(x_0) + \lambda_0 \cdot g(x_0) = f(x_0)$ достигается и локальный минимум $f(x)$. Предположим теперь что в окрестности x_0 есть другая точка x_1, для которой $g(x_1) \leqslant 0$. Но тогда так как $g(x_0) = 0$, то в направлении от x_0 к x_1 (формулировка подходит для многомерного случая с частными производными) $g'(x) \leqslant 0$. А значит $f'(x) \geqslant 0$. Таким образом, мы никак не сможем уменьшить $f(x)$, а значит в стационарной точке и вправду будет локальный минимум $f(x)$ с выполнением ограничения $g(x) \leqslant 0$. Теперь рассмотрим двойственную задачу минимизации.

$$minimize\ f(x),\ with\ constraints\ f_k(x) \leqslant 0$$

$$L(x, \lambda) = f(x) + \sum_k \lambda_k f_k(x)$$

Итак, мы записали Лагранжиан, теперь воспользуемся тем, что в его минимуме будет достигаться минимум $f(x)$ с данными ограничениями. Введём двойственную функцию Лагранжа:

$$g(\lambda) = inf_{x \in D}\, L(x, \lambda)$$

Здесь D - множество значений векторов x, для точного решения необходимо, чтобы $D \in \mathbb{R}^n$ было выпуклым подмножеством, в случае нашей задачи это выполняется. Теперь воспользуемся тем, что $L(x, \lambda) \leqslant f(x)$ в точках минимума. А значит $g(\lambda) \leqslant L(x, \lambda) \leqslant f(x)$. Теперь мы готовы к осознанию двойственной задачи Лагранжа [7]. Мы получили функцию $g(\lambda)$, которая никак не зависит от первоначальных x, потому что их минимизирует. Эта функция в точках оптимума равна Лагранжиану, который в свою очередь $\leqslant f(x)$. Итак, чтобы оба неравенства превратить в равенство, нам нужно максимизировать $g(\lambda)$. Это-то и будет двойственной задачей Лагранжа. Конечно, мы тратим немало усилий на нахождение самой двойственной функции, но зато у нас больше нет ограничений, как в первоначальной задаче. Перейдём обратно к методу опорных векторов.
Лагранжиан с коэффициентами α для нашей задачи имеет такой вид:

$$L(w, b, \alpha) = \lambda \cdot ||w||^2 - \sum_i \alpha_i (y_i (w^T \cdot x_i + b)) - 1)$$

Продифференцируем по ∂w и ∂b:

$$\frac{\partial L(w, b, \alpha)}{\partial w} = w - \sum_i \alpha_i y_i x_i = 0 \ \rightarrow \ w = \sum_i \alpha_i y_i x_i$$

$$\frac{\partial L(w,b,\alpha)}{\partial b} = \sum_i \alpha_i y_i = 0$$

Подставив эти выражения в Лагранжиан, получаем двойственную функцию:

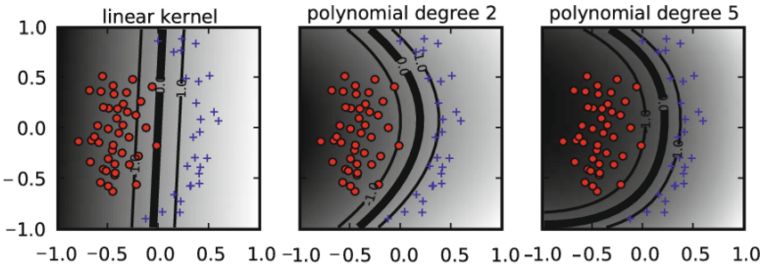
$$g(\alpha) = \sum_i a_i - \lambda \cdot \sum_i \alpha_i \alpha_j y_i y_j (x_i^T \cdot x_j)$$

Нам нужно максимизировать эту функцию с учётом второго ограничения из частных производных. Вот такую долгую дорогу мы прошли, но оно того стоило. Заметим, что теперь все пары x_i и x_j легко выделяются. Заменим x_i на $\phi(x_i)$, тогда $x_i^T \cdot x_j$ можно заменить на $K(x_i, x_j) = \phi(x_i)^T \cdot \phi(x_j)$. Таким образом с помощью выбора ϕ мы можем отображать вектора в пространства большей размерности, разделяя их гиперплоскостями в них.
Сама задача оптимизации двойственной функции Лагранжа решается методами квадратичного программирования, но это уже совсем другая степь. Как правило, используются функции по типу scipy.optimize.minimize, которые сами выбирают лучший способ решения задачи. Теперь выпишем формулы непосредственно для w и b, а также для ответа на тестовые данные. Мы уже знаем, что для w действительна формула из ограничения.

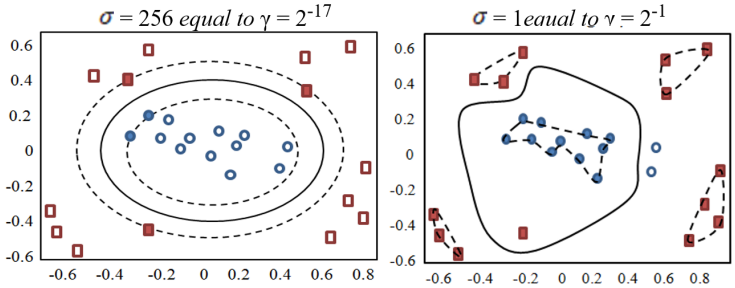
$$w = \sum_i \alpha_i y_i \phi(x_i) \quad b = y_i - w^T \cdot \phi(x_i) = y_i - \sum_j \alpha_j y_j \cdot K(x_i, x_j) \quad x' \to sign(w^T \cdot x' + b)$$

Остаётся чуть подробнее поговорить про ядра [3], после чего перейти к описанию своего собственного кода и проделанной работы. Я перечислю наиболее популярные виды нелинейных ядер, покажу примеры их работы.

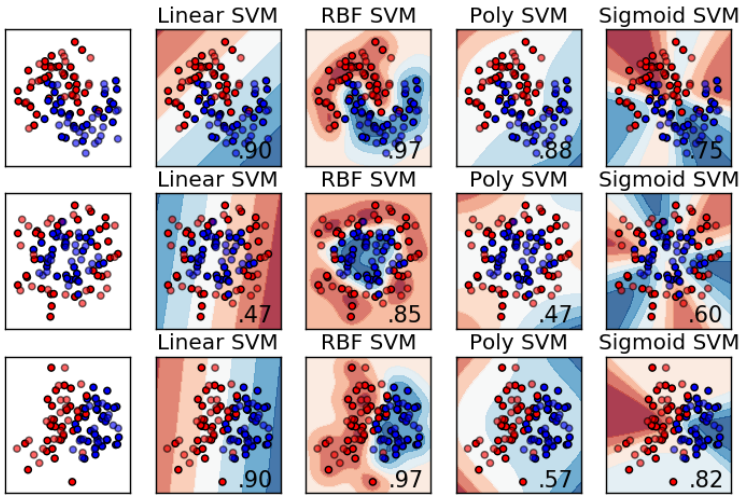
1. Polynomial Kernel: $K(x_i, x_j) = (x_i \cdot x_j)^d$. Мы отображаем наше пространство с заданным скалярным произведением в новое, дополняя его измерениями степеней от 1 до d. В случае линейного ядра мы по факту берём $d = 1$. Представим, что будет при $dim(x_i) = 2$ и $d = 2$: Двумерная картинка разбросанных x_i переходит в трёхмерное пространство, где по третьей оси располагаются квадраты x_i.



2. Radial Basis Function Kernel: $K(x_i, x_j) = exp(-\frac{||x_i - x_j||^2}{\sigma^2})$. Часто встречается в формате $K(x_i, x_j) = exp(-\gamma||x_i - x_j||^2)$, $\gamma = 1/2\sigma^2$. Зачастую rbf-kernel можно интерпретировать как "меру похожести двух векторов так как его действие похоже на нормальное распределение. Но не стоит забывать, что работаем мы с векторами, поэтому по факту строится отображение в бесконечномерное пространство (исходя из определения многомерной экспоненты).



3. Sigmoid Function Kernel: $K(x_i, x_j) = tanh(k(x_i \cdot x_j) + c)$, $k > 0, c < 0$. Это ядро зачастую используется, потому что многие функции распределения случайных величин похожи на сигмоидные функции.



Теперь поговорим про другой вариант решения задачи оптимизации SVM'а, который я использовал. Это своего рода вариация Стохастического Градиентного Спуска [10]. Для начала давайте поймём, почему обычный градиентный спуск не работает. Вспомним, что $\frac{\partial L}{\partial b} = \sum_i \alpha_i y_i = 0$. Если мы будем спускаться за раз лишь по одной координате, то мы нарушим это ограничение, ведь сумма изменится только в одном слагаемом. Тогда нам в голову может придти идея - а давайте выбирать не одну координату, а две: α_i, α_j. Тогда мы будем решать нашу задачу с учётом ограничения $\Delta\alpha_i \cdot y_i + \Delta\alpha_j \cdot y_j = 0$. Именно то и делает алгоритм оптимизации: рандомно выбирает несколько индексов, по которым делает очередной шаг градиентного спуска.

3 Результаты

Основным результатом проекта для себя я считаю погружение в проблематику обнаружения изменений на спутниковых данных, а также вполне глубокое изучение непосредственно алгоритма классификации на Опорных Векторах. Я прочитал достаточно много статей по обеим темам, изучил немалую математическую базу алгоритма; получил своего рода первый опыт самостоятельного исследования какого-то метода машинного обучения и последующего применения его на практике.

Теперь давайте перейдём к практическим результатам проекта (ведь он всё-таки программный). Основная реализация была описана ранее, весь код лежит на github.

3.1 Первая программа

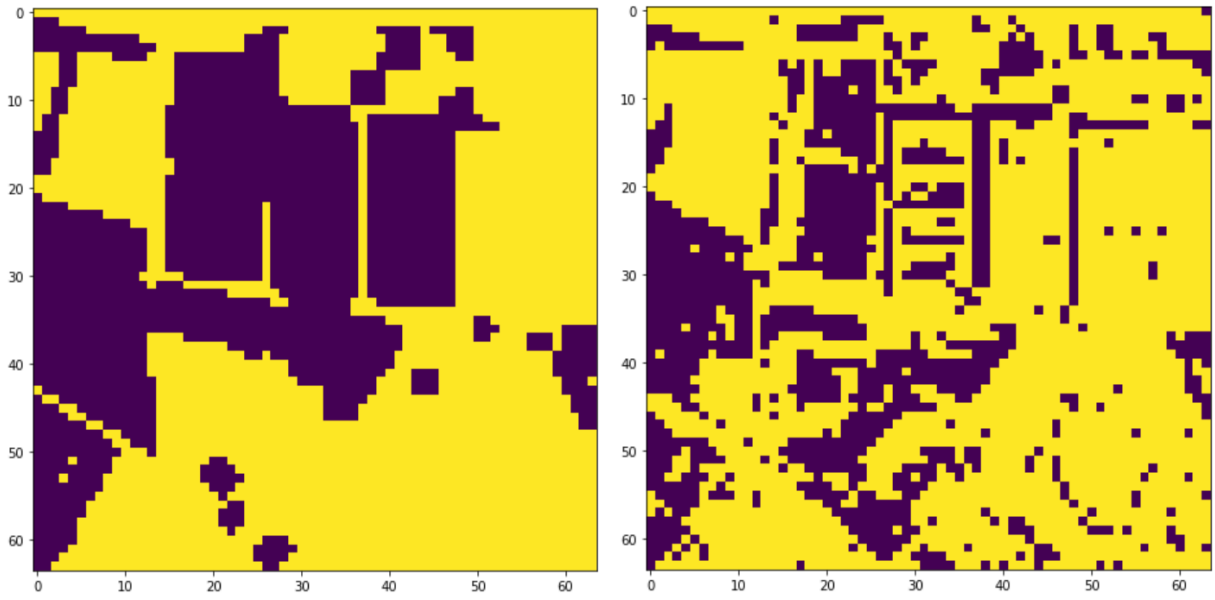
Рассмотрим результаты по первой программе. Мне удалось получить идентичные результаты по accuracy_score между моим SVM'ом с использованием SGD и sklearn.svm.SVC, однако скорость решения задачи оказалась удручающей, чего и следовало ожидать. Мой алгоритм обучался примерно в 43 раза медленнее - и тому есть несколько объяснений: sklearn.svm.SVC использует библотеку libsvm, написанную на Си, что уже существенно повышает производительность метода; внутри себя алгоритм сам определяет лучший способ решения задачи, вызывает наиболее выгодные методы оптимизации, автоматически подстраивает некоторые параметры. И всё же я доволен результатом по этому направлению, так как реализация алгоритма вышла не просто "сухой"и принесла мне новый опыт - я попрактиковался с подбором лучших параметров, подготовкой данных с отбросом ненужных параметров, поработал с дебагом алгоритма.

3.2 Вторая программа

Основную сложность здесь составляла правильная обработка данных, я отбрасывал несколько методов, которые не показали себя эффективными в данной задаче. Выбор наиболее эффективных параметров играл ключевую роль, во многом засчёт ограниченности в железе. Например, даже с кластеризацией обучение модели на 100 изображениях шло 10 часов и не закончилось, потому что время обработки росло экспоненциально. Всё же вот параметры, которые я счёл оптимальными при тренировке алгоритма:

- RBF kernel, проявил себя лучше Linear, Polynomial и Sigmoid
- gamma = 0.3 для RBF kernel
- C = 0.1, параметр trade-off'а между точностью и выбросами
- Использование изображений в чб, дабы сократить время обучения алгоритма. Вынужденная мера, но иначе алгоритм на моём компьютере работает десятки часов.
- Кластеризация с кластерами по 8 либо по 16 пикселей (изображения 512 * 512).

Пример работы алгоритма после обучения на 32-х изображениях с применением кластеризации 8 * 8 пикселей (сначала выводятся правильная классификация, потом предсказанная, изменённые пиксели жёлтые):



Не стоит забывать, что модель обучалась на малом числе изображений с отбросом многих характеристик из-за ограниченных мощностей.
Для наглядности вот время работы алгортима на разных train_set'ах:

- 10 чб изображений 512 * 512, кластеры по 8 * 8 пикселей: fit занимает 2.5 минуты
- 16 чб изображений 512 * 512, кластеры по 8 * 8 пикселей: fit занимает 5.5 минуты
- 32 чб изображений 512 * 512, кластеры по 8 * 8 пикселей: fit занимает 58 минут

Список литературы

[1] Dataset second. URL: https://paperswithcode.com/dataset/second.

[2] Deep learning for change detection in remote sensing images: Comprehensive review and meta-analysis. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9136674.

[3] Different svm kernels testing.

[4] Expectation maximization wikipedia page. URL: https://en.wikipedia.org/wiki/Expectation%E2%80%93maximization_algorithm.

[5] Hinge loss wikipedia page. URL: https://en.wikipedia.org/wiki/Hinge_loss.

[6] Kernel trick wikipedia page. URL: https://ru.wikipedia.org/wiki/%D0%AF%D0%B4%D0%B5%D1%80%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4.

[7] Lagrange multiplier wikipedia page. URL: https://en.wikipedia.org/wiki/Lagrange_multiplier.

[8] sklearn.svm. URL: https://scikit-learn.org/stable/modules/svm.html.

[9] Support vector machines wikipedia page. URL: https://en.wikipedia.org/wiki/Support-vector_machine.

[10] Two-variable dual coordinate descent methods for linear svm. URL: https://www.csie.ntu.edu.tw/~cjlin/papers/2var_cd/twocddual.pdf.

[11] Masroor Hussain. Change detection from remotely sensed images: From pixel-based to object-based approaches. 2013.