Contents

למידת מכונה 5

14 במאי 2020

1 תרגיל 1 :

הגדרנו את $h_{\mathcal{D}}(x)=\begin{cases}1 & \mathcal{P}(y=1|x)\geq\frac{1}{2}\\ -1 & otherwise\end{cases}$. ניזכר כי לפי כלל בייס $\mathcal{P}(x|y)\mathcal{P}(y)=\mathcal{P}(y|x)\mathcal{P}(x)$ מצד שני מאחר ו $\sum\mathcal{P}(y|x)=1$ ו- $y\in\{-1,1\}$ אז מתקיים כי אם $\mathcal{P}(y=1|x)\geq\frac{1}{2}$ אז- $\mathcal{P}(y=1|x)=\max_{y\in\{-1,1\}}\{\mathcal{P}(y|x)\}$. באותו אופן עבור עבור $\mathcal{P}(y=-1|x)\geq\frac{1}{2}$.
ולכן :

$$
\begin{aligned}
h_{\mathcal{D}}(x)&=\arg\max_{y\in\{-1,1\}}\mathcal{P}(y|x)=\mathcal{P}(x)\arg\max_{y\in\{-1,1\}}\mathcal{P}(y|x)\overset{(1)}{=}\\
&=\arg\max_{y\in\{-1,1\}}\mathcal{P}(y|x)\mathcal{P}(x)=\arg\max_{y\in\{-1,1\}}\mathcal{P}(x|y)\mathcal{P}(y)
\end{aligned}
$$

כאשר הכנסנו את x מאחר והוא מגדיל את כל האיברים מהם אנו מחלצים את המקסימום באופן זהה.

2 תרגיל 2:

ממונטוניות חזקה של פונקציית ה $\ln$ נקבל כי $\arg\max_y g(y)=\arg\max_y\ln(g(y))$. ולכן

$$
\begin{aligned}
\Rightarrow h_{\mathcal{D}}(x)&=\arg\max_{y\in\{-1,1\}}\mathcal{P}(x|y)\mathcal{P}(y)=\arg\max_{y\in\{-1,1\}}\ln(\mathcal{P}(x|y)\mathcal{P}(y))=\\
&=\arg\max_{y\in\{-1,1\}}\left\{\ln\left(\frac{1}{\sqrt{(2\pi)^d\det(\Sigma)}}e^{-\frac{1}{2}(x-\mu_y)^T\Sigma^{-1}(x-\mu_y)}\right)+\ln(\mathcal{P}(y))\right\}=\\
&=\arg\max_{y\in\{-1,1\}}\left\{\underbrace{\ln\left(\frac{1}{\sqrt{(2\pi)^d\det(\Sigma)}}\right)}_{not\,depend\,at\,y}+\ln\left(e^{-\frac{1}{2}(x-\mu_y)^T\Sigma^{-1}(x-\mu_y)}\right)+\ln(\mathcal{P}(y))\right\}=\\
&=\arg\max_{y\in\{-1,1\}}\left\{\ln\left(e^{-\frac{1}{2}(x-\mu_y)^T\Sigma^{-1}(x-\mu_y)}\right)+\ln(\mathcal{P}(y))\right\}=\arg\max_{y\in\{-1,1\}}\left\{-\frac{1}{2}(x-\mu_y)^T\Sigma^{-1}(x-\mu_y)+\ln(\mathcal{P}(y))\right\}=\\
&\arg\max_{y\in\{-1,1\}}\left\{\underbrace{-\frac{1}{2}x^T\Sigma^{-1}x}_{not\,depend\,at\,y}+x^T\Sigma^{-1}\mu_y-\frac{1}{2}\mu_y^T\Sigma^{-1}\mu_y+\ln(\mathcal{P}(y))\right\}=\\
&\arg\max_{y\in\{-1,1\}}\left\{x^T\Sigma^{-1}\mu_y-\frac{1}{2}\mu_y^T\Sigma^{-1}\mu_y+\ln(\mathcal{P}(y))\right\}=\arg\max_{y\in\{-1,1\}}\delta_y
\end{aligned}
$$

3 תרגיל 3:

את Σ ו $\mathcal{P}$ נחשב על ידי החישוב הרגיל על ה$training-samples$. (חישוב ישיר) מה שמשתנה ולא היה בשאלות עד כה זה החישוב של μ_+ ו μ_- אבל גם כאן זה מאוד פשוט. $\mu_\pm=Mean\{x|(x,y)=(x,\pm1)\}$.

4 תרגיל 4:

נירצה שהבחינה של הודעה נורמטיבת כהודעת ספאם תיהיה : $Type-2-error$. ולכן נתייג כ $False\,Positive.=\{not-spam,true\}$ ואת : $True\,Positive.=\{spam,\,false\}$.

5 תרגיל 5:

אני מצרף תמונה בכתב יד מאחר וזה הרבה יותר ברור ככה :

SVM - Formulation 5

(נגדיר את $Q = 2I$ ואת $a = 0$ ונראה כי:

$$\forall_i \quad y_i \cancel{\text{sign}} \cdot (\langle \omega, x_i \rangle + b) \geq 1$$

$$\iff \quad y_i \langle \omega, x_i \rangle \geq 1 - b y_i$$

$$\Rightarrow \begin{bmatrix} - & y_1 & - \\ - & y_2 & - \\ & \vdots & \\ - & y_m & - \end{bmatrix} \begin{bmatrix} - & \omega & - \\ - & \omega & - \\ & \vdots & \\ - & \omega & - \end{bmatrix} \begin{bmatrix} | & | & & | \\ x_1 & x_2 & \cdots & x_m \\ | & | & & | \end{bmatrix} \geq \begin{bmatrix} 1 - b y_1 \\ \vdots \\ 1 - b y_m \end{bmatrix}$$

$$\Rightarrow \underbrace{- \begin{bmatrix} - & y_1 & - \\ & \vdots & \\ - & y_i & - \end{bmatrix} \begin{bmatrix} - & \omega & - \\ & \vdots & \\ - & \omega & - \end{bmatrix}}_{A = -y^T \mathbb{1}_{m \times m} \omega} \underbrace{\begin{bmatrix} | & & | \\ x_1 & \cdots & x_m \\ | & & | \end{bmatrix}}_{v} \leq \underbrace{\begin{bmatrix} 1 - b y_1 \\ \vdots \\ 1 - b y_m \end{bmatrix}}_{d}$$

מטריצה של 1-ים

6 תרגיל 6:

נראה שקילות בין שתי הבעיות :

$$\min_{\omega, \xi_i, y_i \langle \omega, x_i \rangle \geq 1 - \xi_i} \frac{\lambda}{2} ||\omega||^2 + \frac{1}{m} \sum_{i=1}^{m} \xi_i = \min_{\omega, \omega, \xi_i, y_i \langle \omega, x_i \rangle \geq 1 - \xi_i} \left\{ \frac{\lambda}{2} ||\omega||^2 + \frac{1}{m} \sum_{i=1}^{m} \min_{\xi_i} \xi_i \right\} =$$

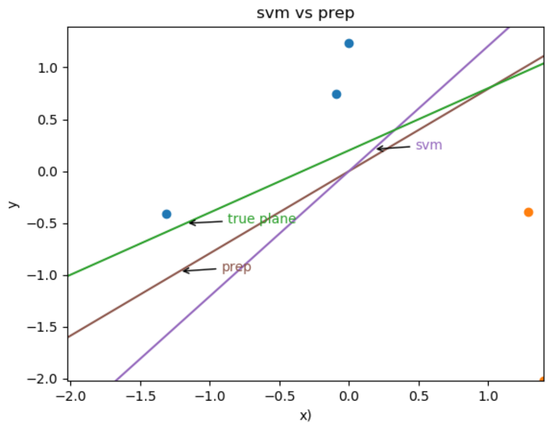
$$= \min \left\{ \frac{\lambda}{2} ||\omega||^2 + \frac{1}{m} \sum_{i=1}^{m} \begin{cases} 1 - y_i \langle \omega, x_i \rangle & if \ 1 - y_i \langle \omega, x_i \rangle \geq 0 \\ 0 & otherwise \end{cases} \right\} =$$

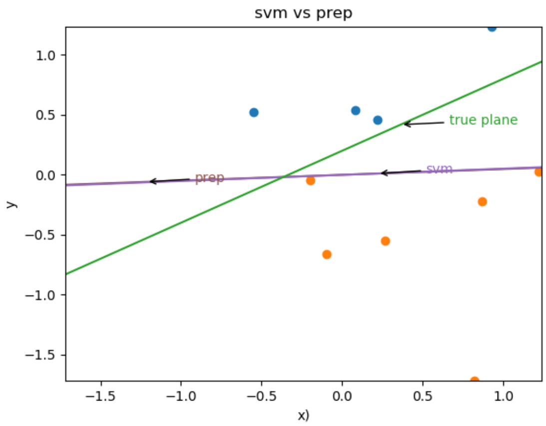
$$= \min_{\omega} \left\{ \frac{\lambda}{2} ||\omega||^2 + \frac{1}{m} \sum_{i=1}^{m} \max \{0, 1 - y_i \langle \omega, x_i \rangle\} \right\}$$

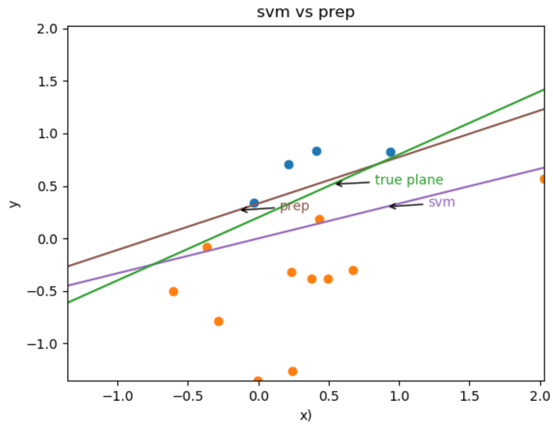
נשים לב שכל מעבר הוא בשיוון ולכן הבכרח פתרון של בעיה אחת פותר גם את הבעיה השניה.

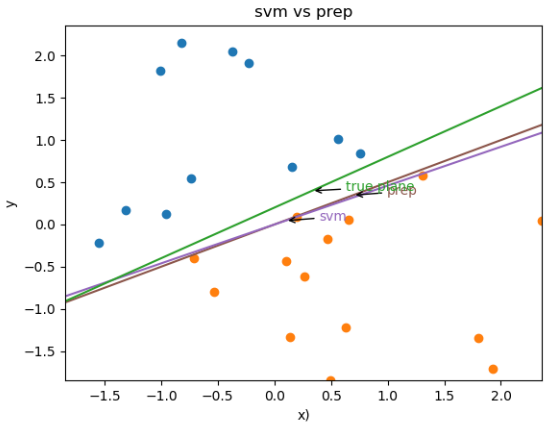
7 השוואה בין שיטות למציאת ישר מפריד.

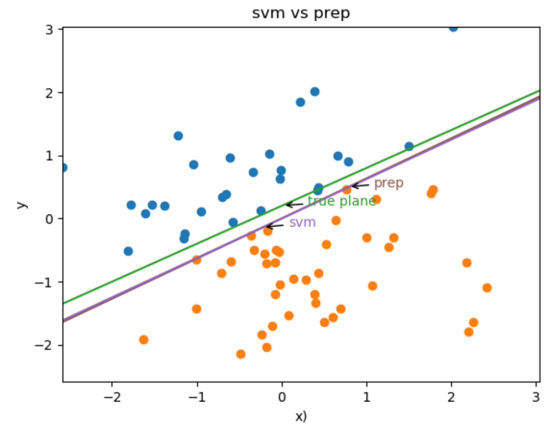
כל השיטות נמצאו יעילות מבחינת יכולת הדיוק, נציין ש LDA איטית באופן משמעותי, בעיקר השלב של מציאת המטריצה ההופכית ($\Sigma^{\dagger}$). הגרפים של $preception$ לעומת svm . ניתן ממש לראות איך המישורים מתאחדים :





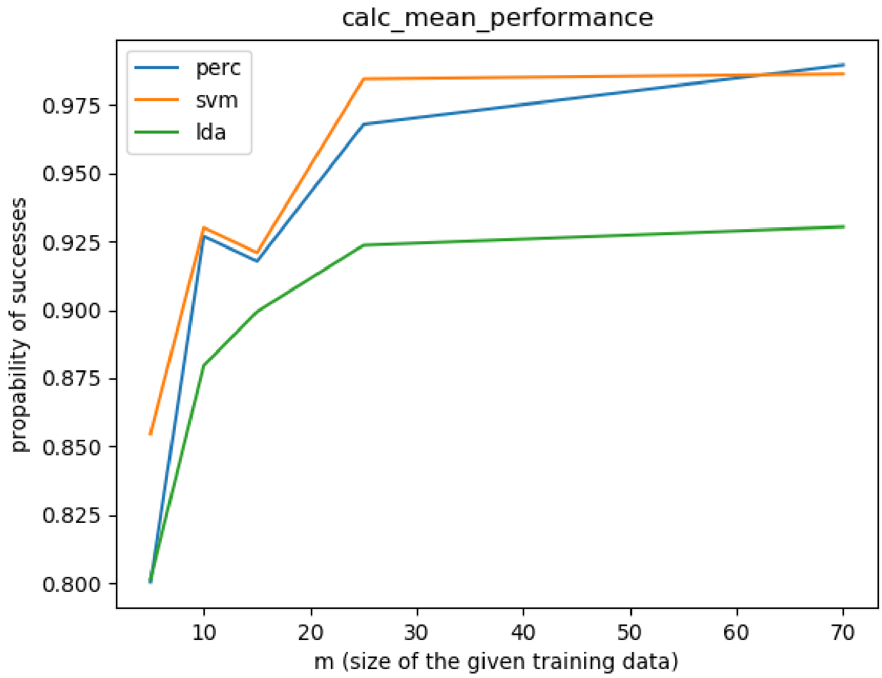






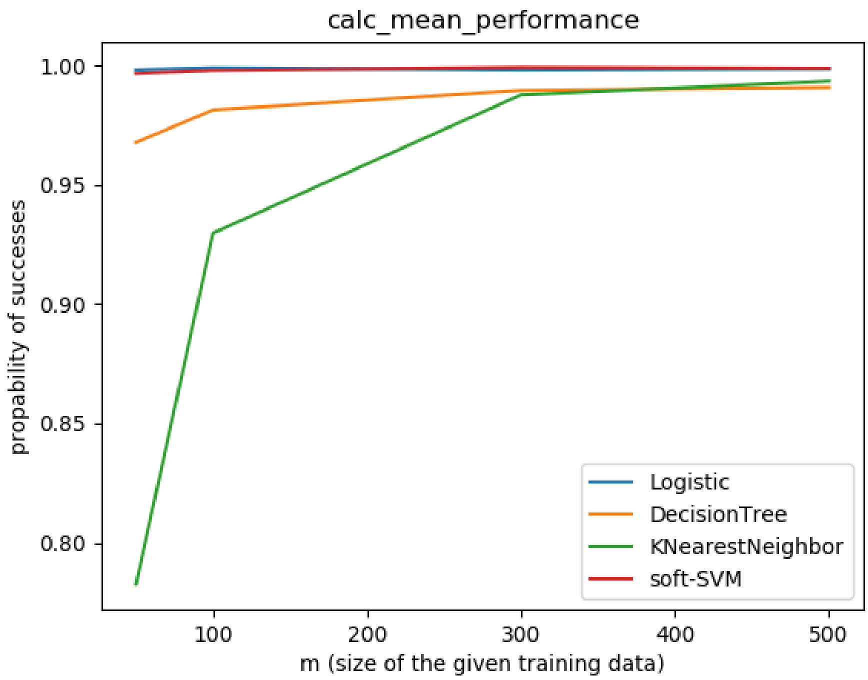
7.1 ההשוואה בין בשיטות השונות לLDA.

קל ליראות ש LDA לא נותן לנו שיערוך טוב. (כלומר כן טוב אבל פחות) אני מאמין שהוא סובל יותר מאובר פיט. מאחר ומראש הוא לא בהכרח מייצג מישור ולכן ספיציפית לבעייה הזאת הוא פחות מתאים.



8 זיהוי ספרות :

ההשוואה על זיהוי הספרות, אני חייב להגיש שממש הואשמתי מי מידת הדיוק. שיטת ה SVM לקחה את רוב זמן החישוב כאשר על הקלטים הגדולים זמן הריצה היה באזור $0.05\ [sec] \sim$. כל שאר השיטות ירדו מ 0.025 שניות עבור הקלטים הגדולים.



2 ./comparsion.py

```
import numpy as np
import pandas as pd
from sklearn.svm import SVC

from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D # <--- This is important for 3d plotting
import pandas as pd
from pandas import DataFrame
from plotnine import *

import matplotlib as mpl
import matplotlib.pyplot as plt

from random import random

from models import SVM, LDA, Perceptron


def label_f(weight, bias):
    def _label_f(x):
        return np.sign( np.dot(weight, x) + bias)
    return _label_f

def draw_points(m, label_function=label_f(np.array([0.3, -0.5]), 0.1)):
    def _draw_points_n(m,n):
        return np.random.multivariate_normal(
         np.zeros(n), np.identity(n), m)

    X = _draw_points_n(m, 2)
    y = np.array([ label_function(vec) for vec in X ])
    return X,y

def analyze_clssifiers( ):


    # def generate_line_prec(prec):

    def plot(prec, _svm, X, y):
        plt.scatter( X[y == -1][:,0], X[y == -1][:,1])
        plt.scatter( X[y ==  1][:,0], X[y ==  1][:,1])

        min_x, max_x = min(X[:,0]), max(X[:,0])
        min_y, max_y = min(X[:,1]), max(X[:,1])
        print(min_x, max_x)

        _min_range, _max_range = min(min_x, min_y) , max(max_x, max_y)
        print(prec.W)
        xx = [_min_range, _max_range]

        def get_y(W, _x):
            return -(W[0] + _x * W[1])/ W[2] if W[2]  != 0 else -W[0]

        def get_y_prep(_x):
            return get_y(prec.W, _x )

```

```
        def get_y_svm(_x):
            print(_svm.coef_()[0])
            return get_y(_svm.coef_()[0], _x)

        def get_true_y(_x):
            return 0.1/0.5 + 0.3/0.5 * _x

        plt.xlim([_min_range,_max_range])
        plt.ylim([_min_range,_max_range])



        middle = (_min_range + _max_range) /2

        def print_line(msg, _f, _color):
            xx = [_min_range , _max_range ]
            yy = [ _f(_x) for _x in xx  ]
            _x = middle + 2 * (0.5 - random())
            _y =_f(_x)
            plt.plot(xx, yy, color=_color )
            plt.annotate(msg, color=_color,
                  xy=(_x, _y), xycoords='data',
                  xytext=(_x + 0.3 , _y), textcoords='data',
                  arrowprops=dict(arrowstyle="->"))

        print_line("prep", get_y_prep, "C5")
        print_line("svm" , get_y_svm , "C4")
        print_line("true plane" , get_true_y ,"C2")
        plt.title("svm vs prep")
        plt.xlabel("x)")
        plt.ylabel("y")
        plt.show()

    for m in [5, 10, 15, 25, 70]:
        X, y = draw_points(m)
        blues, reds = X[y==1], X[y==-1]
        _modes = [Perceptron(), SVM()]
        for _model in _modes:
            _model.fit(deepcopy(X),y)

        plot( _modes[0], _modes[1], X, y)


def expanded_analyze_clssifiers():
    times, k = 500, 1000
    modles = []
    models_num = 3


    def genrate_real_plane(m):
        _f = label_f(np.array([random(), random()]), random())
        X, y = draw_points(m, label_function=_f )
        while (-1 not in y) or (1 not in y):
             X, y = draw_points(m, label_function= _f)
        return X, y, _f

    def accur(mod, _f, Z):
        _prob = 0
        for x,y in zip(map(_f, Z), mod.predict(Z)):
            if x == y:
                _prob +=1
        return _prob/len(Z)

    def one_iteraion(m):
        _modes = [Perceptron(), SVM(), LDA()]
        X, y, _f = genrate_real_plane(m)
        ret = []
        for _model in _modes:
```

```python
            _model.fit(deepcopy(X),y)
            Z, _ = draw_points(k)
            ret.append( accur(_model, _f, Z) )
        return np.array(ret)

    def calc_mean_performance(M = [5, 10, 15, 25, 70] ):
        ret = []
        for m in M:
            _mean = np.zeros(models_num)
            for _ in range(times):
                _mean += one_iteraion(m)
            ret.append( _mean/ times )
        return M, np.array( ret )

    m, mean_performance = calc_mean_performance()
    for _model_num, _name  in enumerate(["perc", "svm", "lda"]):
        print(mean_performance)
        plt.plot( m , mean_performance[:,_model_num] )
    plt.legend( ["perc", "svm", "lda"] )
    plt.title("calc_mean_performance")
    plt.xlabel("m (size of the given training data)")
    plt.ylabel("propability of successes")
    plt.show()

if __name__ == "__main__" :
    X,y = draw_points(10)
    #p = Perceptron()


    from copy import deepcopy

    models_class = [ Perceptron, SVM, Logistic, DecisionTree, LDA ]
    models = [ ]
    for mod in models_class:
        models.append( mod( ) )
        print("{} init ".format( type(mod) ))

    for mod in models:
        mod.fit(deepcopy(X),y)
        print("{} fit ".format( type(mod) ))

    for mod in models:
        print(mod.predict(deepcopy(X)))
        print("{} predict ".format( type(mod) ))

    analyze_clssifiers()
    expanded_analyze_clssifiers()
```

3 ./mnsit data.py

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

from models import Logistic, DecisionTree, KNearestNeighbor, SVM
from copy import deepcopy

import time

def load_data():
    mnist = tf.keras.datasets.mnist
    (x_train, y_train),(x_test, y_test) = mnist.load_data()
    train_images = np.logical_or((y_train == 0), (y_train == 1))
    test_images = np.logical_or((y_test == 0), (y_test == 1))
    x_train, y_train = x_train[train_images], y_train[train_images]
    x_test, y_test = x_test[test_images], y_test[test_images]

    return x_train, y_train, x_test, y_test


def rearrange_data(X):
    return np.array( [ x.flatten() for x in X ])


def expanded_analyze_clssifiers(x_train, y_train, x_test, y_test):
    times, k = 50, 100
    modles = []
    models_num = 4

    def generate( m, X, y ):
        indexs = [False] * len(X)
        _indexs = np.random.choice( list(range(len(X)) ), size=m)
        for _index in _indexs:
            indexs[_index] = True

        return X[indexs], y[indexs], indexs

    def accur(mod, indexs , Z):
        _prob = 0
        for x,y in zip(y_test[indexs], mod.predict(Z)):
            if x == y:
                _prob +=1
        return _prob/len(Z)

    def one_iteraion(m):
        _modes = [Logistic(), DecisionTree(), KNearestNeighbor(), SVM()]
        X, y, indexs = generate(m, x_train, y_train)

        # your code


        while (0 not in y) or (1 not in y) :
             X, y, indexs = generate(m, x_train, y_train)
        ret = []
        for _model in _modes:
            start_time = time.time()
            _model.fit(deepcopy(X),y)
            elapsed_time = time.time() - start_time
```

```
            print("train : {} takes {}".format(_model, elapsed_time))
            Z, _, indexs = generate(k, x_test, y_test)
            ret.append( accur(_model, indexs, Z) )
        return np.array(ret)

    def calc_mean_performance(M = [50, 100, 300, 500] ):
        ret = []
        for m in M:
            _mean = np.zeros(models_num)
            for _ in range(times):

                _mean += one_iteraion(m)
            ret.append( _mean/ times )
        return M, np.array( ret )

    m, mean_performance = calc_mean_performance()
    for _model_num, _name  in enumerate(["Logistic", "DecisionTree", "KNearestNeighbor", "soft-SVM"]):
        plt.plot( m , mean_performance[:,_model_num] )
    plt.legend( ["Logistic", "DecisionTree", "KNearestNeighbor", "soft-SVM"] )
    plt.title("calc_mean_performance")
    plt.xlabel("m (size of the given training data)")
    plt.ylabel("propability of successes")
    plt.show()


if __name__ == '__main__':
    x_train, y_train, x_test, y_test = load_data()
    expanded_analyze_clssifiers(rearrange_data(x_train),
     rearrange_data(y_train), rearrange_data(x_test), rearrange_data(y_test))
```

4 ./models.py

```python
import numpy as np
import pandas as pd
from sklearn.svm import SVC

from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D # <--- This is important for 3d plotting
import pandas as pd
from pandas import DataFrame
from plotnine import *

import matplotlib as mpl
import matplotlib.pyplot as plt

from random import random

class abcModel:

    def __init__(self):
        self.mod = None

    def fit(self, X, y):
        self.mod.fit(self.bias(X), y )

    def predict(self, X):
        return self.mod.predict(self.bias(X))

    def get_hyperplan(self):
        pass

    def bias(self, X):
        return np.insert(X,0, 1, 1)

    def draw(self):
        pass

    def score(self, X, y):
        return {
                num_samples : 0,
                error : 0,
                accuracy :0,
                FPR : 0,
                TPR : 0,
                precision : 0,
                recall : 0
            }

# [v]
class Perceptron(abcModel):
    def __init__(self):
        self.W = np.array( [] )
        super().__init__()

    def fit(self, _X, y):
        X = self.bias(_X)

```

```
        self.W = np.zeros( shape = X.shape[1] )
        def not_classifiy():
            return [ (x,y) for x in X if self.predict(x) != y ]

        _updated = True
        while _updated:
            _updated = False
            for i in range(len(y)):
                if np.dot(self.W, X[i]) * y[i] <= 0:
                    self.W = self.W + (X[i]* y[i])
                    _updated = True
        return self.W

    def predict(self, X):
        return np.sign(self.W @ self.bias(X).transpose())

class LDA(abcModel):
    def __init__(self):
        super().__init__()


    def fit (self, X, y):
        X = self.bias(X)

        def gen_delta_y(X ,y ,y_val):
            _X = X[ y == y_val  ]
            lnP = np.log( len( y == y_val  ) / len(y)  )

            aritmetic_mean = np.array( [np.mean( u ) for u in _X.transpose()] )
            _cov = np.cov( X.transpose() )
            _inv_cov = np.linalg.pinv(_cov)

            def delta(x):
                return x.transpose() @ _inv_cov @  aritmetic_mean -\
                 0.5*aritmetic_mean.transpose() @ _inv_cov @  aritmetic_mean +\
                 lnP


            return delta

        self.deltas = [ gen_delta_y(X ,y ,y_val) for y_val in [-1, 1] ]

    def predict(self, U):
        return np.array([{ 0 : -1.0, 1 : 1.0 }[np.argmax( [_delta(u) for _delta in self.deltas] ) ] for u in self.bias(U) ])


class SVM(abcModel):
    def __init__(self):
        super().__init__()
        self.mod = SVC(C=1e10, kernel='linear')

    def coef_(self):
        return self.mod.coef_

class Logistic(abcModel):
    def __init__(self):
        super().__init__()
        self.mod = LogisticRegression(solver='liblinear')

class DecisionTree(abcModel):
    def __init__(self):
        super().__init__()
        self.mod = DecisionTreeClassifier()

class KNearestNeighbor(abcModel):

    def __init__ (self):
        super().__init__()
```

```
        self.mod =  KNeighborsClassifier(n_neighbors=40)



def label_f(weight, bias):
    def _label_f(x):
        return np.sign( np.dot(weight, x) + bias)
    return _label_f

def draw_points(m, label_function=label_f(np.array([0.3, -0.5]), 0.1)):
    def _draw_points_n(m,n):
        return np.random.multivariate_normal(
         np.zeros(n), np.identity(n), m)

    X = _draw_points_n(m, 2)
    y = np.array([ label_function(vec) for vec in X ])
    return X,y

def analyze_clssifiers( ):


    # def generate_line_prec(prec):

    def plot(prec, _svm, X, y):
        plt.scatter( X[y == -1][:,0], X[y == -1][:,1])
        plt.scatter( X[y ==  1][:,0], X[y ==  1][:,1])

        min_x, max_x = min(X[:,0]), max(X[:,0])
        min_y, max_y = min(X[:,1]), max(X[:,1])

        _min_range, _max_range = min(min_x, min_y) , max(max_x, max_y)
        xx = [_min_range, _max_range]

        def get_y(W, _x):
            return -(W[0] + _x * W[1])/ W[2] if W[2]  != 0 else -W[0]

        def get_y_prep(_x):
            return get_y(prec.W, _x )

        def get_y_svm(_x):
            print(_svm.coef_()[0])
            return get_y(_svm.coef_()[0], _x)

        def get_true_y(_x):
            return 0.1/0.5 + 0.3/0.5 * _x

        plt.xlim([_min_range,_max_range])
        plt.ylim([_min_range,_max_range])



        middle = (_min_range + _max_range) /2

        def print_line(msg, _f, _color):
            xx = [_min_range , _max_range ]
            yy = [ _f(_x) for _x in xx  ]
            _x = middle + 2 * (0.5 - random())
            _y =_f(_x)
            plt.plot(xx, yy, color=_color )
            plt.annotate(msg, color=_color,
                  xy=(_x, _y), xycoords='data',
                  xytext=(_x + 0.3 , _y), textcoords='data',
                  arrowprops=dict(arrowstyle="->"))

        print_line("prep", get_y_prep, "C5")
        print_line("svm" , get_y_svm , "C4")
        print_line("true plane" , get_true_y ,"C2")
        plt.title("svm vs prep")
```

```python
        plt.xlabel("x)")
        plt.ylabel("y")
        plt.show()

    for m in [5, 10, 15, 25, 70]:
        X, y = draw_points(m)
        blues, reds = X[y==1], X[y==-1]
        _modes = [Perceptron(), SVM()]
        for _model in _modes:
            _model.fit(deepcopy(X),y)

        plot( _modes[0], _modes[1], X, y)


def expanded_analyze_clssifiers():
    times, k = 7, 1000
    modles = []
    models_num = 3


    def genrate_real_plane(m):
        _f = label_f(np.array([random(), random()]), random())
        X, y = draw_points(m, label_function=_f )
        while (-1 not in y) or (1 not in y):
             X, y = draw_points(m, label_function= _f)
        return X, y, _f

    def accur(mod, _f, Z):
        _prob = 0
        for x,y in zip(map(_f, Z), mod.predict(Z)):
            if x == y:
                _prob +=1
        return _prob/len(Z)

    def one_iteraion(m):
        _modes = [Perceptron(), SVM(), LDA()]
        X, y, _f = genrate_real_plane(m)
        ret = []
        for _model in _modes:
            print(type(_model))
            _model.fit(deepcopy(X),y)
            Z, _ = draw_points(k)
            ret.append( accur(_model, _f, Z) )
        return np.array(ret)

    def calc_mean_performance(M = [5, 10, 15, 25, 70] ):
        ret = []
        for m in M:
            _mean = np.zeros(models_num)
            for _ in range(times):
                _mean += one_iteraion(m)
            ret.append( _mean/ times )
        return M, np.array( ret )

    m, mean_performance = calc_mean_performance()
    for _model_num, _name  in enumerate(["perc", "svm", "lda"]):
        print(_name)
        print(mean_performance)
        plt.plot( m , mean_performance[:,_model_num] )
    plt.legend( ["perc", "svm", "lda"] )
    plt.title("calc_mean_performance")
    plt.xlabel("m (size of the given training data)")
    plt.ylabel("propability of successes")
    plt.show()

if __name__ == "__main__" :
    X,y = draw_points(10)
    #p = Perceptron()
```

```


from copy import deepcopy

models_class = [ Perceptron, SVM, Logistic, DecisionTree, LDA ]
models = [ ]
for mod in models_class:
    models.append( mod( ) )
    print("{} init ".format( type(mod) ))

for mod in models:
    mod.fit(deepcopy(X),y)
    print("{} fit ".format( type(mod) ))

for mod in models:
    print(mod.predict(deepcopy(X)))
    print("{} predict ".format( type(mod) ))

#analyze_clssifiers()
expanded_analyze_clssifiers()
```