IE 7300: Statistical Learning for Engineering

HW 8

Youyu Zhang

zhang.youy@northeastern.edu

(530)574-2826

Code available on: https://github.com/kuohu233/IE_7300

Submitted by 11/15/2022

In [74]:
```
## imports ##
import numpy as np
import pandas as pd
from scipy import stats
import matplotlib.pyplot as plt
import seaborn as sns
from typing import Dict, Any
from abc import ABC,abstractmethod
from sklearn.preprocessing import StandardScaler
```

Create custom classification models using the Bank Marketing dataset, https://archive-beta.ics.uci.edu/dataset/222/bank+marketing Links to an external site., and evaluate your model results. Split the dataset into training and test datasets 80:20.

In [143…
```
df = pd.read_csv('bank.csv')
```

In [144…
```
# Setup column names
column_names = []
for i in str(list(df.columns)).split(';'):
    column_names.append(i[1:-1])
column_names[0] = 'age'
column_names[-1] = 'y'
```

In [145…
```
# Split and add data
data_dict = {}
for i in column_names:
    data_dict[i] = []

for i in range(df.shape[0]):
    item = df.iloc[i,0].split(';')
    for j in range(len(item)):
        data_dict[column_names[j]].append(item[j])

data = pd.DataFrame(data=data_dict, columns=column_names)

# Convert type of each column
convert_dict = {'age':int, 'job':str, 'marital':str, 'education':str,
                'default':str, 'balance':int, 'housing':str, 'loan':str,
                'contact':str, 'day':int, 'month':str, 'duration':int,
                'campaign':int, 'pdays':int,'previous':int,
                'poutcome':str,'y':str}
data = data.astype(convert_dict)

data.default = pd.Series(np.where(data.default.values == '"yes"', 1, 0), data.index)
data.housing = pd.Series(np.where(data.housing.values == '"yes"', 1, 0), data.index)
data.loan = pd.Series(np.where(data.loan.values == '"yes"', 1, 0), data.index)
data.y = pd.Series(np.where(data.y.values == '"yes"', 1, 0), data.index)
```

```python
# Dummy
cat_vars = ['job','marital','education','contact','month', 'poutcome']

for j in cat_vars:
    var = []
    for i in range(data.shape[0]):
        var.append(data[j][i][1:-1])
    data[j] = var

for j in cat_vars:
    dummies = pd.get_dummies(data[j],prefix=j)
    data = pd.concat([data, dummies], axis='columns')
    data = data.drop([j], axis='columns')

data.head()
```

Out[145]:

	age	default	balance	housing	loan	day	duration	campaign	pdays	previous	...	month_jun	month_mar
0	30	0	1787	0	0	19	79	1	-1	0	...	0	0
1	33	0	4789	1	1	11	220	1	339	4	...	0	0
2	35	0	1350	1	0	16	185	1	330	1	...	0	0
3	30	0	1476	1	1	3	199	4	-1	0	...	1	0
4	59	0	0	1	0	5	226	1	-1	0	...	0	0

5 rows × 49 columns

In [146...
```python
data.shape
```

Out[146]:
```
(4521, 49)
```

In [147...
```python
# Separate into training and testing dataset.
x_all = data.drop(['y'], axis=1)
y_all = data['y']
x_train, x_test = np.split(x_all,[int(0.8*len(x_all))])
y_train, y_test = np.split(y_all,[int(0.8*len(y_all))])
```

(B)

Create an SVM and Knn model. Fit the model using the training dataset, and find the model accuracy and confusion matrix. Explain each model's outcome and accuracy. (10 points)

KNN

In [148...
```python
class KNN(ABC):
    """
    Base class for KNN implementations
    """

    def __init__(self, K : int = 3, metric : str = 'minkowski', p : int = 2) -> None:
        """
        Initializer function. Ensure that input parameters are compatiable.
        Inputs:
            K      -> integer specifying number of neighbours to consider
            metric -> string to indicate the distance metric to use (valid entries are '
            p      -> order of the minkowski metric (valid only when distance == 'minkow
        """
        # check distance is a valid entry
        valid_distance = ['minkowski','cosine']
```

```
        if metric not in valid_distance:
            msg = "Entered value for metric is not valid. Pick one of {}".format(valid_d
            raise ValueError(msg)
        # check minkowski p parameter
        if (metric == 'minkowski') and (p <= 0):
            msg = "Entered value for p is not valid. For metric = 'minkowski', p >= 1"
            raise ValueError(msg)
        # store/initialise input parameters
        self.K       = K
        self.metric  = metric
        self.p       = p
        self.X_train = np.array([])
        self.y_train = np.array([])

    def __del__(self) -> None:
        """
        Destructor function.
        """
        del self.K
        del self.metric
        del self.p
        del self.X_train
        del self.y_train

    def __minkowski(self, x : np.array) -> np.array:
        """
        Private function to compute the minkowski distance between point x and the train
        Inputs:
            x -> numpy data point of predictors to consider
        Outputs:
            np.array -> numpy array of the computed distances
        """
        return np.power(np.sum(np.power(np.abs(self.X_train - x),self.p),axis=1),1/self.

    def __cosine(self, x : np.array) -> np.array:
        """
        Private function to compute the cosine distance between point x and the training
        Inputs:
            x -> numpy data point of predictors to consider
        Outputs:
            np.array -> numpy array of the computed distances
        """
        return (1 - (np.dot(self.X_train,x)/(np.linalg.norm(x)*np.linalg.norm(self.X_tra

    def __distances(self, X : np.array) -> np.array:
        """
        Private function to compute distances to each point x in X[x,:]
        Inputs:
            X -> numpy array of points [x]
        Outputs:
            D -> numpy array containing distances from x to all points in the training s
        """
        # cover distance calculation
        if self.metric == 'minkowski':
            D = np.apply_along_axis(self.__minkowski,1,X)
        elif self.metric == 'cosine':
            D = np.apply_along_axis(self.__cosine,1,X)
        # return computed distances
        return D

    @abstractmethod
    def _generate_predictions(self, idx_neighbours : np.array) -> np.array:
        """
        Protected function to compute predictions from the K nearest neighbours
        """
        pass
```

```python
    def fit(self, X : np.array, y : np.array) -> None:
        """
        Public training function for the class. It is assummed input X has been normalis
        Inputs:
            X -> numpy array containing the predictor features
            y -> numpy array containing the labels associated with each value in X
        """
        # store training data
        self.X_train = np.copy(X)
        self.y_train = np.copy(y)

    def predict(self, X : np.array) -> np.array:
        """
        Public prediction function for the class.
        It is assummed input X has been normalised in the same fashion as the input to t
        Inputs:
            X -> numpy array containing the predictor features
        Outputs:
           y_pred -> numpy array containing the predicted labels
        """
        # ensure we have already trained the instance
        if (self.X_train.size == 0) or (self.y_train.size == 0):
            raise Exception('Model is not trained. Call fit before calling predict.')
        # compute distances
        D = self.__distances(X)
        # obtain indices for the K nearest neighbours
        idx_neighbours = D.argsort()[:,:self.K]
        # compute predictions
        y_pred = self._generate_predictions(idx_neighbours)
        # return results
        return y_pred

    def get_params(self, deep : bool = False) -> Dict:
        """
        Public function to return model parameters
        Inputs:
            deep -> boolean input parameter
        Outputs:
            Dict -> dictionary of stored class input parameters
        """
        return {'K':self.K,
                'metric':self.metric,
                'p':self.p}


class KNNClassifier(KNN):
    """
    Class for KNN classifiction implementation
    """

    def __init__(self, K : int = 3, metric : str = 'minkowski', p : int = 2) -> None:
        """
        Initializer function. Ensure that input parameters are compatiable.
        Inputs:
            K       -> integer specifying number of neighbours to consider
            metric  -> string to indicate the distance metric to use (valid entries are
            p       -> order of the minkowski metric (valid only when distance == 'minko
        """
        # call base class initialiser
        super().__init__(K,metric,p)

    def _generate_predictions(self, idx_neighbours : np.array) -> np.array:
        """
        Protected function to compute predictions from the K nearest neighbours
        Inputs:
```

```
            idx_neighbours -> indices of nearest neighbours
        Outputs:
            y_pred -> numpy array of prediction results
        """
        # compute the mode label for each submitted sample
        y_pred = stats.mode(self.y_train[idx_neighbours],axis=1).mode.flatten()
        # return result
        return y_pred
```

In [149…

```
knn = KNNClassifier()
knn.fit(np.array(x_train),np.array(y_train))
y_pred_knn_train = knn.predict(np.array(x_train))
```

Confusion matrix below inidicates that most of the cases are correctly classified. The accuracy of the model in training dataset is 92.1%.

The prediction accuracy was not good when y=yes (row 1). More than 50% of the prediction was false to be y=no. This could happen because of the unbalanced training data sample.

In [150…

```
from sklearn.metrics import confusion_matrix

confusion_matrix(y_true=np.array(y_train), y_pred=y_pred_knn_train)
```

Out[150]:

```
array([[3140,   66],
       [ 221,  189]], dtype=int64)
```

In [151…

```
def accuracy(y_pred, y_true):
    return round(np.sum(y_pred==y_true)/len(y_true),4)

accuracy(y_pred=y_pred_knn_train, y_true=y_train)
```

Out[151]:

```
0.9206
```

SVM

In [152…

```
class SVM:
    def __init__(self, learning_rate=0.001, lambda_param=0.01, n_iters=1000):
        self.lr = learning_rate
        self.lambda_param = lambda_param
        self.n_iters = n_iters
        self.w = None
        self.b = None

    def fit(self, X, y):
        n_samples, n_features = X.shape

        y_ = np.where(y <= 0, -1, 1)

        self.w = np.zeros(n_features)
        self.b = 0

        for _ in range(self.n_iters):
            for idx, x_i in enumerate(X):
                condition = y_[idx] * (np.dot(x_i, self.w) - self.b) >= 1
                if condition:
                    self.w -= self.lr * (2 * self.lambda_param * self.w)
                else:
                    self.w -= self.lr * (
                        2 * self.lambda_param * self.w - np.dot(x_i, y_[idx])
                    )
                    self.b -= self.lr * y_[idx]
```

```
    def predict(self, X):
        approx = np.dot(X, self.w) - self.b
        return np.sign(approx)
```

In [153…
```
svm = SVM()
svm.fit(np.array(x_train),np.array(y_train))
y_pred_svm_train = svm.predict(np.array(x_train))
```

The overall accuracy of SVM is 88.1%, a little bit lower than that of KNN. The confusion matrix below indicate the misclassification cases of y=yes is twice more than that of y=no.

In [154…
```
result_train_svm = pd.DataFrame(data={'train':y_train, 'pred':pd.Series(np.where(y_pred_
confusion_matrix(y_true=np.array(y_train), y_pred=result_train_svm['pred'])
```

Out[154]:
```
array([[3161,   45],
       [ 384,   26]], dtype=int64)
```

In [156…
```
accuracy(y_pred=result_train_svm['pred'], y_true=y_train)
```

Out[156]:
```
0.8814
```

(C)

Compare the Knn model with a few K values and find the best k model. Describe each model performance (10 points)

The accuracy result indicates that k=3 will give highest accuracy in training dataset. The confusion matrix also prove that k=3 will give least misclassification in cases where y=yes. As k increases over 3, the accuracy drops slowly, and thus I would recommend a model with k=3.

For k=2, the prediction on cases where y=no (row 0) reaches 100% correct. Meanwhile the cases where y=yes was not fully predicted.

For k=4, the misclassification cases increases on both conditions, and similarly to k=5. The overall accuracy of the two models are not compatible to k=3.

In [160…
```
test_k = [2, 3, 4, 5]
test_acc = []
for i in test_k:
    knn_test = KNNClassifier(i)
    knn_test.fit(np.array(x_train),np.array(y_train))
    y_pred_knn_train_test = knn_test.predict(np.array(x_train))
    test_acc.append(accuracy(y_pred=y_pred_knn_train_test, y_true=y_train))
    print(confusion_matrix(y_true=np.array(y_train), y_pred=y_pred_knn_train_test))
```

```
[[3206    0]
 [ 298  112]]
[[3140   66]
 [ 221  189]]
[[3183   23]
 [ 316   94]]
[[3144   62]
 [ 274  136]]
```

In [161…
```
test_acc
```

Out[161]:
```
[0.9176, 0.9206, 0.9062, 0.9071]
```

Compare your SVM model with various kernel methods (linear, rbf, and polynomial). Explain each Kernal parameter (12 points)

In [173…

```
from scipy import optimize

class KernelSvmClassifier:

    def __init__(self, C, kernel):
        self.C = C
        self.kernel = kernel          # <---
        self.alpha = None
        self.supportVectors = None

    def fit(self, X, y):
        N = len(y)
        # --->
        # Gram matrix of h(x) y
        hXX = np.apply_along_axis(lambda x1 : np.apply_along_axis(lambda x2:  self.kerne
                                  1, X)
        yp = y.reshape(-1, 1)
        GramHXy = hXX * np.matmul(yp, yp.T)
        # <---

        # Lagrange dual problem
        def Ld0(G, alpha):
            return alpha.sum() - 0.5 * alpha.dot(alpha.dot(G))

        # Partial derivate of Ld on alpha
        def Ld0dAlpha(G, alpha):
            return np.ones_like(alpha) - alpha.dot(G)

        # Constraints on alpha of the shape :
        # -  d - C*alpha  = 0
        # -  b - A*alpha >= 0
        A = np.vstack((-np.eye(N), np.eye(N)))             # <---
        b = np.hstack((np.zeros(N), self.C * np.ones(N)))  # <---
        constraints = ({'type': 'eq',   'fun': lambda a: np.dot(a, y),     'jac': lambda
                       {'type': 'ineq', 'fun': lambda a: b - np.dot(A, a), 'jac': lambda

        # Maximize by minimizing the opposite
        optRes = optimize.minimize(fun=lambda a: -Ld0(GramHXy, a),
                                   x0=np.ones(N),
                                   method='SLSQP',
                                   jac=lambda a: -Ld0dAlpha(GramHXy, a),
                                   constraints=constraints)
        self.alpha = optRes.x
        # --->
        epsilon = 1e-8
        supportIndices = self.alpha > epsilon
        self.supportVectors = X[supportIndices]
        self.supportAlphaY = y[supportIndices] * self.alpha[supportIndices]
        # <---

    def predict(self, X):
        """ Predict y values in {-1, 1} """
        # --->
        def predict1(x):
            x1 = np.apply_along_axis(lambda s: self.kernel(s, x), 1, self.supportVectors
            x2 = x1 * self.supportAlphaY
            return np.sum(x2)

        d = np.apply_along_axis(predict1, 1, X)
        return 2 * (d > 0) - 1
        # <---

def GRBF(x1, x2):
```

```
    diff = x1 - x2
    return np.exp(-np.dot(diff, diff) * len(x1) / 2)

def poly(x1, x2):
    return np.dot(x1, x2)
```

The result of grbf kernel in SVM doesn't look good. The overall accuracy is 88.7%, which is lower than all the previous knn models. All the y-yes cases are misclassified which means this model doesn't work well on this condition. It also take much longer time in fitting than other models.

In [163…
```
svm_grbf = KernelSvmClassifier(C=1, kernel=GRBF)
svm_grbf.fit(np.array(x_train), np.array(y_train))
```

```
C:\Users\youyu\AppData\Local\Temp/ipykernel_9672/1102196909.py:65: RuntimeWarning: overf
low encountered in int_scalars
  return np.exp(-np.dot(diff, diff) * len(x1) / 2)
C:\Users\youyu\AppData\Local\Temp/ipykernel_9672/1102196909.py:65: RuntimeWarning: overf
low encountered in exp
  return np.exp(-np.dot(diff, diff) * len(x1) / 2)
C:\Users\youyu\AppData\Local\Temp/ipykernel_9672/1102196909.py:18: RuntimeWarning: inval
id value encountered in multiply
  GramHXy = hXX * np.matmul(yp, yp.T)
```

In [165…
```
y_pred_svmgrbf_train = svm_grbf.predict(np.array(x_train))
```

```
C:\Users\youyu\AppData\Local\Temp/ipykernel_9672/1102196909.py:65: RuntimeWarning: overf
low encountered in int_scalars
  return np.exp(-np.dot(diff, diff) * len(x1) / 2)
C:\Users\youyu\AppData\Local\Temp/ipykernel_9672/1102196909.py:65: RuntimeWarning: overf
low encountered in exp
  return np.exp(-np.dot(diff, diff) * len(x1) / 2)
C:\Users\youyu\AppData\Local\Temp/ipykernel_9672/1102196909.py:56: RuntimeWarning: inval
id value encountered in multiply
  x2 = x1 * self.supportAlphaY
```

In [167…
```
result_train_svmgrbf = pd.DataFrame(data={'train':y_train, 'pred':pd.Series(np.where(y_p
confusion_matrix(y_true=np.array(y_train), y_pred=result_train_svmgrbf['pred'])
```

Out[167]:
```
array([[3206,    0],
       [ 410,    0]], dtype=int64)
```

In [168…
```
accuracy(y_pred=result_train_svmgrbf['pred'],y_true=np.array(y_train))
```

Out[168]:
```
0.8866
```

The polynomial kernel looks to have the same prediction as grbf got before. The reason is that the polynomial was not working or the kernel function is not correctly used.

In [203…
```
svm_poly = KernelSvmClassifier(C=1, kernel=poly)
svm_poly.fit(np.array(x_train), np.array(y_train))
```

In [204…
```
y_pred_svmpoly_train = svm_poly.predict(np.array(x_train))
```

In [205…
```
result_train_svmpoly = pd.DataFrame(data={'train':y_train, 'pred':pd.Series(np.where(y_p
confusion_matrix(y_true=np.array(y_train), y_pred=result_train_svmpoly['pred'])
```

Out[205]:
```
array([[3206,    0],
       [ 410,    0]], dtype=int64)
```

In [206…
```
accuracy(y_pred=result_train_svmpoly['pred'],y_true=np.array(y_train))
```

Out[206]:
```
0.8866
```

The accuracy of SVM in linear kernel looks similar to grbf, but the misclassification cases distribution is not. Both conditions have misclassification, and evenly distributed corresponding to their proportions.

In [170…
```
result_train_svm = pd.DataFrame(data={'train':y_train, 'pred':pd.Series(np.where(y_pred_
confusion_matrix(y_true=np.array(y_train), y_pred=result_train_svm['pred'])
```

Out[170]:
```
array([[3161,   45],
       [ 384,   26]], dtype=int64)
```

In [171…
```
accuracy(y_pred=result_train_svm['pred'], y_true=y_train)
```

Out[171]:
```
0.8814
```

(D)

Analyze your SVM and Knn model performance with outlier and imbalanced samples (make a sample from the given dataset).

How do the SVM and Knn models handle the outliers and imbalanced datasets with statistical evidence? (10 points)

In [189…
```
# Data sampling
subset1 = data[data['y']==1][0:10]
subset2 = data[data['y']==0][0:10]
subset = pd.concat([subset1, subset2])

x_sample = subset.drop(['y'], axis=1)
y_sample = subset['y']
```

In [190…
```
sample_acc = []
knn_sample = KNNClassifier(3)
knn_sample.fit(np.array(x_sample),np.array(y_sample))
y_pred_knn_train_sample = knn_sample.predict(np.array(x_sample))
sample_acc.append(accuracy(y_pred=y_pred_knn_train_sample, y_true=y_sample))
```

In order to balance the previous data, I subset the original dataset and chose samples evenly based on the situations of Y. A small sample dataset of 20 cases were used here. Both y=no and y=yes have 10 cases.

The confusion matrix and accuracy below indicated much lower efficiency in this knn model. The misclassification of both conditions are the same and both are lower than previous model. The overall accuracy is 80% and it can be worse in test dataset instead of training dataset. Larger dataset is needed for KNN model in order to provide better prediction and accuracy.

In KNN, prediction of a single case is made by the nearby k cases. In order to get the nearby k points, distances are calculated, and the outliers means the points that are far away from the other points, and thus the accuracy of the outliers will drop. If the data subset is chosen based on the x_train similarity, the accuracy can be higher in the training dataset (the accuracy of testing dataset cannot be guaranteed because the dataset size is smaller).

In [191…
```
print(confusion_matrix(y_true=np.array(y_sample), y_pred=y_pred_knn_train_sample))
```

```
[[8 2]
 [2 8]]
```

In [192…
```
print(sample_acc)
```

```
[0.8]
```

Similarly to SVM model. The chosen small sample cannot reach the accuracy in the previous trials. The confusion matrix cannot easily tell if the prediction is correct or wrong because all the situations are alike. That means a large partion of the cases are wrongly classified. The accuracy of 70% is the lowest among all the models until now.

The outliers are not that important to SVM compared to KNN because SVM is designed to minimized the distances from the hyperplane to the data points of each group meanwhile maximize the distances between the groups. In the current subset, the distance beween groups are not large enough to maximize the soft margin, and thus the result and accuracy looks not good.

In [193…
```
svm_sample = SVM()
svm_sample.fit(np.array(x_sample),np.array(y_sample))
y_pred_svm_sample = svm_sample.predict(np.array(x_sample))
```

In [196…
```
result_sample_svm = pd.DataFrame(data={'train':np.array(y_sample), 'pred':pd.Series(np.w
confusion_matrix(y_true=np.array(y_sample), y_pred=result_sample_svm['pred'])
```

Out[196]:
```
array([[8, 2],
       [4, 6]], dtype=int64)
```

In [207…
```
accuracy(y_true=np.array(y_sample), y_pred=result_sample_svm['pred'])
```

Out[207]:
```
0.7
```