

به نام خدا



دانشکده مهندسی کامپیوتر

دانشگاه صنعتی امیرکبیر

دانشکده مهندسی کامپیوتر

استاد درس: دکتر امین ناظرفرد

زمستان ۱۳۹۹

درس یادگیری ماشین

سروش مهدی

شماره دانشجویی: ۹۹۱۳۱۰۵۰

پروژه نهایی

**فهرست مطالب**

۱		۱
۱	۱.۱
۷	۲.۱
۷	۱.۲.۱
۷	۲.۲.۱
۸	۳.۲.۱
۹		۲
۱۲		۳
۱۲	۱.۳
۱۵	۲.۳
۱۸		۴
۱۸	۱.۴
۱۸	۲.۴
۲۱	۳.۴
۲۲	۴.۴
۲۲	۵.۴
۲۲		۵
۲۲	۱.۵
۲۴	۲.۵
۲۵	۳.۵
۲۶	۴.۵



۱.۱

در این بخش برای خوشه بندی از کتابخانه SKlearn استفاده شده است. کد های این بخش به صورت زیر است.

```
#reading images
bee_img=mpimg.imread('bee.jpg')
parrots_img=mpimg.imread('parrots.jpg')

for img , name in [(bee_img , 'bee') , (parrots_img , 'parrots')]: 

    #normalizing image
    train_img = np.array(img, dtype=np.float64) / 255
    w , h , d = img.shape
    #vectorizing image
    train_img = np.reshape(train_img, (w * h, d))

    for k in [2,3,4,5,6,10,15,20]:
        #performing kmeans
        kmeans = KMeans(n_clusters=k, random_state=0).fit(train_img)
        labels = kmeans.predict(train_img)
        new_img = recreate_image(kmeans.cluster_centers_ , labels, w, h)
        mpimg.imsave('k='+str(k)+name +'.jpg' , new_img)
```

Listing : ۱

در این قسمت ابتدا تصاویر را خوانده و سپس به ازای مقادیر مختلف K الگوریتم Kmeans را اجرا میکنیم و در نهایت تصویر را ذخیره میکنیم.
برای خوشه بندی پیکسل ها ماتریس تصویر را که هر عنصر ان دارای سه مقدار بود به یک وکتور که طول آن برابر طول ضریب عرض تصویر است تبدیل کردیم و سپس خوشه بندی را انجام دادیم. از تابع پایین استفاده میکنیم تا پس از خوشه بندی برچسب های بدست امده برای هر عنصر وکتور را تبدیل به تصویر با طول و عرض تصویر اولیه کنیم

نتایج خواسته شده در صفحات بعد امده اند.

```
def recreate_image(codebook, labels, w, h):
    """Recreate the (compressed) image from the code book & labels

    Args:
        codebook ([numpy array]): [centers of clusters]
        labels ([numpy array]): [label for each pixel in clustering]
        w ([int]): [width of photo]
        h ([int]): [height of photo]

    Returns:
        [numpy array]: [recreated rgb image from labels of
        vectorize image in clustering]
    """
    d = 3
    image = np.zeros((w, h, d))
    label_idx = 0
    for i in range(w):
        for j in range(h):
            #assigning color of center of cluster for each pixel
            image[i][j] = codebook[labels[label_idx]]
            label_idx += 1
    return image
```

Listing :۲



شکل ۱: K=۲



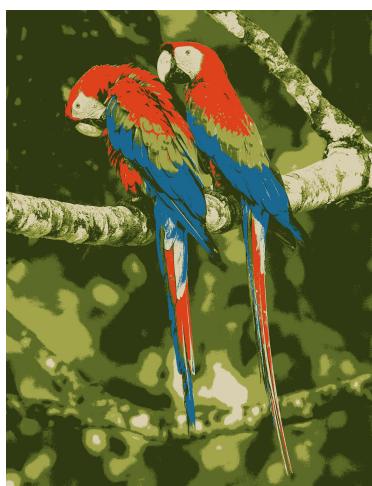
شکل ۲ : K=۳



شکل ۳ : K=۴



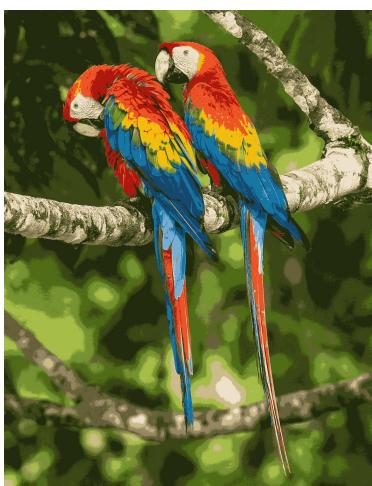
شکل ۴: K=۵



شکل ۵: K=۶



شکل ۱۰: K=۱۰



شکل ۱۵: K=۱۵



شکل ۲۰ : ۸



۲.۱

۱.۲.۱

در این روش ابتدا یک معیار برای تعیین خطای خوش بندی انتخاب میکنیم . سپس به ازای مقادیر مختلف برای تعداد خوش ها این معیار خطرا را محاسبه میکنیم و نمودار این خطاه را بر حسب تعداد خوش ها رسم میکنیم . نمودار به دست امده احتمالا در نقطه ای یک شکستگی دارد که بعد از آن خطاه حدودا به صورت خطی کاهش میابد و نسبت به قبل از این نقطه کاهش چشمگیری ندارد که تعداد خوش های متناظر با این نقطه ای به عنوان تعداد خوش های مدل نهایی انتخاب میکنیم . در واقع میتوان گفت در نمودار بدست امده ما نقطه ای که حدودا متناظر با ارجح نمودار است را به عنوان نقطه بهینه انتخاب میکنیم .

۲.۲.۱

در این قسمت نیز برای خوش بندی از کتابخانه SKlearn استفاده شده است . همچنین برای معیار خطرا ار استفاده شده است . کد این قسمت به صورت زیر است .

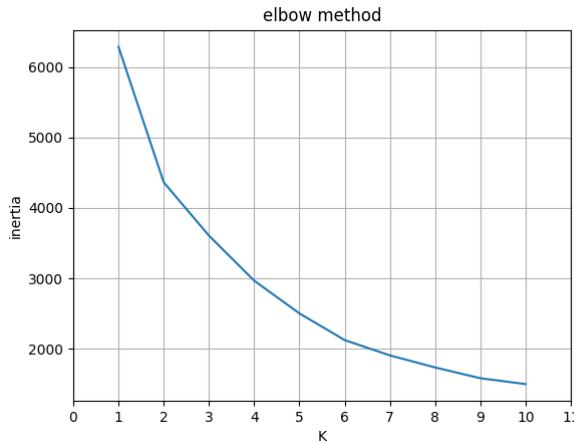
```
def normaliz(col):
    #normalize a feature range
    return (col - col.min())/(col.max() - col.min())

#preprocesssing
#reading data
df = pd.read_csv('Shill Bidding Dataset.csv')
#feature selection
#dropping id and target columns
train_data = df.drop(columns = ['Record_ID' , 'Class' , 'Auction_ID' , 'Bidder_ID' ])
#normalizing Auction_Duration feature
train_data['Auction_Duration'] = normaliz(train_data['Auction_Duration'])
train_data = train_data.to_numpy()

inertia = []
#performing kmeans with different Ks
for k in range(1,11):
    kmeans = KMeans(n_clusters=k, random_state=0).fit(train_data)
    inertia.append(kmeans.inertia_)
```

Listing :۳

برای انتخاب ویژگی ها ستون های ایدی و کلاس را حذف میکنیم . همچنین یکی از ستون ها که نرمال نیست را در بازه صفر و یک نرمال میکنیم .



شکل ۹: تابع هزینه بر حسب تعداد خوشه ها

همانطور که مشاهده میشود میتوان گفت در نقطه K برابر دو شکستگی داریم پس بهترین تعداد خوشه دو میباشد. همینطور اگر به برچسب های واقعی توجه کنیم نیز تعداد کلاس ها دو میباشد.

۳.۲.۱

کد محاسبه purity به صورت زیر است.

```
def purity(g_truth , labels):
    #a function for calculating purity
    #does not works good for imbalanced data
    class_ = []
    #finding must frequent class in each cluster
    for i in range(2):
        zero_num = np.count_nonzero(g_truth[labels==i]==0)
        one_num = np.count_nonzero(g_truth[labels==i]==1)
        if zero_num > one_num:
            class_.append(0)
        else:
            class_.append(1)

    return sum([np.count_nonzero(g_truth[labels==i]==class_[i]) for i in range(2)]) / len(g_truth)
```

Listing : ۴

ورودی اول تابع برچسب واقعی داده ها و ورودی دوم برچسب خوشه ای است که هر داده در ان قرار میگیرد. در این تابع ابتدا در هر خوشه کلاسی که بیشترین تعداد را دارد مشخص میکنیم و به هر خوشه این کلاس را نسبت میدهیم. سپس برای هر خوشه تعداد نقاطی را که به کلاس آن خوشه تعلق دارند محاسبه میکنیم و جمع اعداد بدست امده را بر تعداد کل داده ها تقسیم میکنیم. باید توجه کنیم که این معیار معيار خوبی برای دیتابست های نامتوانن نیست. در اینجا نیز دیتابست ما نامتوانن هست و در نتیجه هنگام محاسبه این معیار



هر دو خوشه یک برچسب میگیرند و این برچسب برچسبی است که بیشتر داده های دیتاست ما از این نوع هستند. در نتیجه جدا از نتیجه خوشه بنده این معیار همیشه برای دیتاست های نامتوازن مقدار بالایی دارد. همچنین این مقدار در این مورد برابر 0.8932 میباشد.

۲

در این سوال برای اجرای الگوریتم از کتابخانه SKlearn استفاده شده است. همچنین پارامتر های الگوریتم نیز با ازمون و خطأ بدست امده اند و بهترین پارامتر ها انتخاب شده اند. از تابع زیر برای محاسبه معیار purity استفاده میشود.

```
def purity(g_truth , labels):
    """a function for calculate purity

Args:
    g_truth ([numpy array]): [real class labels]
    labels ([numpy array]): [cluster labels]

Returns:
    [int]: [purity]
"""

t=0
for i in np.unique(labels):
    if i !=-1:
        counts = np.unique(g_truth[labels==i],return_counts=True)[1]
        t+=np.max(counts)

return t/len(g_truth)
```

Listing :5

همچنین برای اجرای عملیات خواسته شده به ازای هر دیتاست از تابع زیر استفاده میشود. بخش مصورسازی این تابع در گزارش اورده نشده است.
نتیجه اجرای الگوریتم روی دیتاست های داده شده در صفحات بعد امده است.

```

def dbscan_plot(df_name , eps , min_samples):
    """a function for performing dbscan on datasets
       and plotting them it also prints purity for dataset

    Args:
        df_name ([str]): [dataset name]
        eps ([float]): [eps for dbscan]
        min_samples ([int]): [min_samples for dbscan]
    """

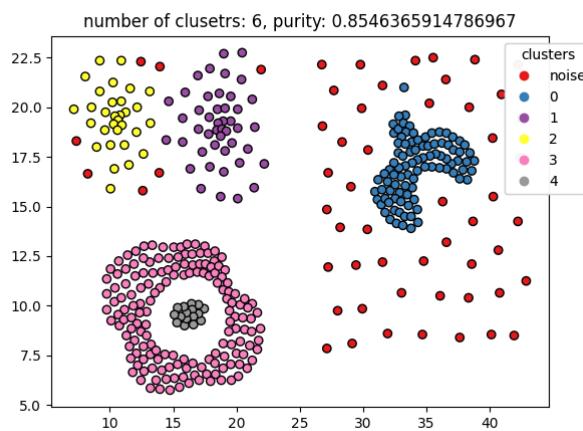
    #fixing column names based on dataset
    if df_name!='rings':
        columns = ['x' , 'y' , 'class']
    else:
        columns = ['class','x' , 'y','z']

    #reading dataset and converting to numpy
    df = pd.read_csv(df_name + '.txt' , sep = '\t' , header=None , names = columns)
    if df_name!='rings':
        train = df[columns[:-1]].to_numpy()
    else:
        train = df[columns[1:]].to_numpy()

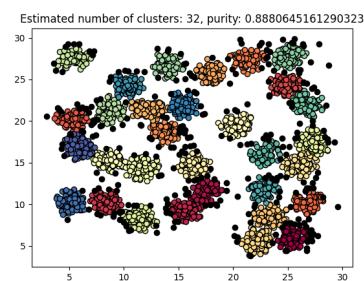
    #performing dbscan
    clustering = DBSCAN(eps = eps , min_samples=min_samples).fit(train)

```

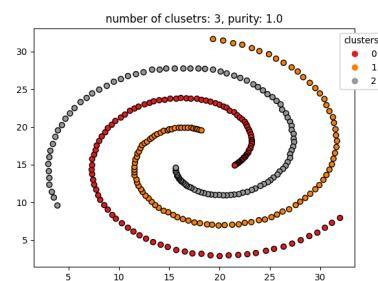
Listing :۸



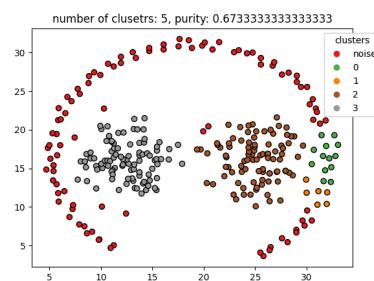
شکل ۱۰: Compound



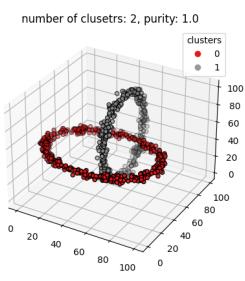
شکل ۱۲ : D۳۱



شکل ۱۱ : spiral



شکل ۱۴ : pathbased



شکل ۱۳ : rings



در مورد دیتا ست های spiral و rings میبینیم که این الگوریتم به خوبی عمل کرده و معیار purity بهینه است. میتوان گفت این الگوریتم در مورد دیتاست هایی که در آن ها خوشه ها چگالی های تقریباً مشابهی دارند و فاصله مناسبی از یکدیگر دارند عملکرد خوبی دارد. در مورد دیتاست compound نیز این موضوع تا حدودی برقرار است. در مورد دیتاست D³¹ چون خوشه ها به یکدیگر نزدیک هستند با توجه های به پارامتر های الگوریتم تعدادی مرزی به عنوان نویز شناسایی میشوند تا الگوریتم عملکرد خوبی داشته باشد. در این دیتاست چگالی خوشه ها مشابه اند اما فاصله خوشه ها با یکدیگر مناسب نیست. در مورد دیتاست pathbased میبینیم بین خوشه ای که به صورت دایره ای شکل دور شکل کشیده شده و دو خوشه دیگر اختلاف چگالی وجود دارد که در این مورد تشخیص خوشه ها برای این الگوریتم مشکل است و عملکرد مناسبی ندارد.

۳

در این قسمت کد ساختن محیط به صورت زیر است.

```
#creating and initializing the map
MAP = [ 'SFFFH',
        'FFHHF',
        'FFFFF',
        'HHFHF',
        'FFFFG']
```

```
env = gym.make('FrozenLake-v0', desc=MAP)
env.reset()
```

Listing : ۷

۱.۳

تابع استفاده شده در این قسمت به صورت زیر هستند اولین تابع برای چاپ سیاست بهینه به فرمت خواسته شده هست.

```
def print_policy(policy):
    #print policy in wanted format
    moves = ['L' , 'D' , 'R' , 'U']
    for i in range(5):
        for j in range(5):
            ind = 5*i + j
            print(moves[int(policy[ind])]) , end ='')
        print()
```

Listing : ۸



تابع بعدی برای یک state به ازای action های مختلف مقدار value را در timestep بعدی محاسبه میکند.

```
def next_step(env, state, V, discount):
    """calculate V in next step for a state for each action

Args:
    env ([gym.env]): [frozen lake environment]
    state ([int]): [number of state]
    V ([numpy array]): [values matrix]
    discount ([float]): [discount rate]

Returns:
    [numpy array]: [value for each action from state]
"""

values = np.zeros(env.nA)
for action in range(env.nA):
    for prob, next_state, reward, done in env.P[state][action]:
        values[action] += prob * (reward + discount * V[next_state])
return values
```

Listing :۹

تابع اصلی برای اجرای الگوریتم در صفحه بعد امده است. دز این تابع ابتدا مقادیر value به ازای همه state ها برابر صفر قرار داده میشود. سپس در یک حلقه برای هر state از بین حرکت هایی که عامل میتواند انجام دهد حرکتی را انتخاب میکند که value را بیشینه کند و این مقدار جدید را برای state value میکند اگر در یک مرحله تغییرات این مقادیر از یک مقدار خاص کوچک تر بود این حلقه متوقف میشود. اگر به تعداد بیشترین تکرار برسد نیز همینظور است. سپس با توجه به مقادیر value بدست امده نهایی سیاست بهینه تعیین میشود. به این صورت که عامل برای هر state حرکتی را انتخاب میکند که مقدار value بیشینه شود. همچنین مقادیر خواسته شده به صورت زیر است.

```
num of iterations for Value-iteration algorithm: 69
V :
[[0.00705818 0.00757541 0.00354976 0.00140339 0.      ],
 [0.01027763 0.01210316 0.          0.          0.10575815],
 [0.0138932 0.02486399 0.06175889 0.09329169 0.2675059 ],
 [0.          0.          0.09981688 0.          0.57087443],
 [0.10128456 0.15490579 0.29053598 0.57997934 0.      ]]

optimal policy:
DLULL
DLLLD
UUDDR
LLLLR
DDDDL
Time: 0.036678300999938074
```

شکل ۱۵: نتایج خواسته شده برای قسمت اول



```
def value_iteration(env, discount=0.85, theta=10**(-9), max_iter=10**9):
    """performing value iteration for

    Args:
        env ([gym.env]): [frozen lake environment]
        discount (float, optional): [discount_factor]. Defaults to 0.85.
        theta ([float], optional): [trashhold for change in values]. Defaults to 1e-10.
        max_iter ([float], optional): [numbet of max iterations for algorithm]. Defaults to 1e9.

    Returns:
        policy [numpy array]: [best policy for each state]
        V [numpy array]: [state value function]
    """
    # set V0 = 0 for all states
    V = np.zeros(env.nS)
    for i in range(max_iter):
        # Early stopping condition
        delta = 0
        # Update each state
        for state in range(env.nS):
            # calculate values for all possible actions in this state
            action_values = next_step(env, state, V, discount)
            # Select best action
            best_action_value = np.max(action_values)
            # Calculate change in value
            delta = max(delta, np.abs(V[state] - best_action_value))
            # Update the value function for current state
            V[state] = best_action_value
            # Check if we can stop
        if delta < theta:
            print('num of iterations for Value-iteration algorithm: ' + str(i))
            break

    # Create a deterministic policy using the optimal value function
    policy = np.zeros([env.nS])
    for state in range(env.nS):
        # One step lookahead to find the best action for this state
        action_value = next_step(env, state, V, discount)
        # Update the policy to perform a better action at a current state
        policy[state] = np.argmax(action_value)

    return policy, V
```

Listing : ۱.

۲.۳

توابع استفاده شده در این قسمت در دو صفحه بعدی آمده اند. تابع اول به ازای یک سیاست معین مقادیر value را محاسبه میکند. تابع دوم الگوریتم خواسته شده را اجرا میکند. به این صورت که ابتدا یک سیاست تصادفی انتخاب میکند که در اینجا ما برای هر خانه احتمال هر حرکت را برابر در نظر گرفتیم سپس در یک حلقه برای این سیاست مقادیر نهایی value را با استفاده از تابع قبلی محاسبه میکند و سپس با توجه به این مقادیر یک سیاست بهتر انتخاب میکند. و این حلقه را تکرار میکند. اگر تعداد تکرار در حلقه به تعداد مشخصی برسد و یا سیاست جدید با قبلی یکی باشد حلقه متوقف میشود.

نتایج خواسته شده در این بخش به صورت زیر هستند.

```

Policy evaluated in 40 iterations.
Policy evaluated in 71 iterations.
Policy evaluated in 71 iterations.
Evaluated 4 policies.
V :
[[0.004213493 0.003171681 0.001403665 0.000378767 0.          ]
 [0.006444166 0.006136718 0.          0.          0.105758147]
 [0.01016323 0.019262828 0.051686504 0.090437848 0.267505901]
 [0.          0.          0.072722279 0.          0.570874426]
 [0.101284557 0.154905795 0.290535984 0.579979343 0.          ]]

optimal policy:
LLULLL
DLLLD
UUDDR
LLLLR
DDDDL
Time: 0.08455954200007909

```

شکل ۱۶: نتایج خواسته شده برای قسمت دوم

مشاهده میشود که زمان اجرا از الگوریتم قبلی بیشتر است. همچنین تعداد تکرار نیز از الگوریتم قبلی بیشتر است اما باید توجه داشت که در هر تکرار این الگوریتم باز محاسباتی کمتری نسبت به تکرار الگوریتم قبلی داریم.



```
def policy_evaluation(policy, env, discount=1.0, theta=10**(-9), max_iter=10**9):
    """a function for calculating V for a policy

Args:
    policy ([numpy array]): [for each state has probability of each action]
    env (gym.env): [frozen lake environment]
    discount (float, optional): [discount rate]. Defaults to 1.0.
    theta ([float], optional): [trashhold for change in values]. Defaults to 10**(-10).
    max_iter ([int], optional): [numbet of max iterations for algorithm]. Defaults to 10**9.

Returns:
    [numpy array]: [state value function for given policy]
"""

evaluation_iterations = 1
# set V0 = 0 for each state
V = np.zeros(env.nS)
# Repeat until change in value is below the threshold
for i in range(int(max_iter)):
    # Initialize a change of value function as zero
    delta = 0

    for state in range(env.nS):

        v = 0
        # Try all possible actions which can be taken from this state
        for action, action_probability in enumerate(policy[state]):
            # Check how good next state will be
            for state_probability, next_state, reward, done in env.P[state][action]:
                # Calculate the expected value
                v += action_probability * state_probability * (reward + discount * V[next_state])

        # Calculate the absolute change of value function
        delta = max(delta, np.abs(V[state] - v))
        # Update value function
        V[state] = v
    evaluation_iterations += 1

    # Terminate if value change is insignificant
    if delta < theta:
        print(f'Policy evaluated in {evaluation_iterations} iterations.')
        return V
```

Listing : ۱۱



```
def policy_iteration(env, discount=.85, max_iter=10**9):
    """performing policy iteration algorithm

    Args:
        env ([gym.env]): [frozen lake environment]
        discount (float, optional): [discount_factor]. Defaults to 0.85.
        max_iter ([float], optional): [numbert of max iterations for algorithm]. Defaults to 1e9.

    Returns:
        policy [numpy array]: [best policy for each state]
        V [numpy array]: [state value function]
    """
    #first policy is a random uniform policy
    policy = np.ones([env.nS, env.nA]) / env.nA

    evaluated_policies = 1

    for i in range(int(max_iter)):
        stable_policy = True
        # Evaluate current policy
        V = policy_evaluation(policy, env, discount=discount)
        #policy Improvement
        for state in range(env.nS):
            # Choose the best action in a current state under current policy
            current_action = np.argmax(policy[state])
            # calculate values for all possible actions in this state
            action_value = next_step(env, state, V, discount)
            # Select best action
            best_action = np.argmax(action_value)
            # If action didn't change
            if current_action != best_action:
                stable_policy = False
                policy[state] = np.eye(env.nA)[best_action]
        evaluated_policies += 1

        if stable_policy:
            print(f'Evaluated {evaluated_policies} policies.')
    return policy, V
```

Listing : ۱۲



۱.۴

کد این بخش به صورت زیر است.

```
#reading dataset
df = pd.read_csv('SeoulBikeData.csv')

target_col = 'Rented Bike Count'
categorical_cols = ['Seasons' , 'Holiday' , 'Functioning Day']
#changing categorical columns to numerical
df[categorical_cols] = df[categorical_cols].astype('category')
df[categorical_cols] = df[categorical_cols].apply(lambda x: x.cat.codes)
#dropping date column
df = df.drop(columns=['Date'])
#selecting target and features
target = df[target_col].to_numpy()
features = df.drop(columns=[target_col])
#normalizing features
for column in features.columns:
    features[column]=(features[column]-features[column].min())/(features[column].max()-features[column].min())
```

Listing : ۱۳

در این کد ابتدا داده های عددي تبدیل میکنیم سپس ویژگی تاریخ را از دیتاست حذف میکنیم چون ارتباط منطقی ای با داده ای که میخواهیم پیش بینی کنیم ندارد. سپس ستون هدف را از ویژگی ها جدا میکنیم و در نهایت داده ها را در بازه صفر و یک نرمال میکنیم.

۲.۴

کد های این قسمت در صفحات بعدی امده اند. تابع `select_features` برای انتخاب ویژگی ها بر اساس ماتریس همبستگی استفاده میشود . این تابع به این صورت عمل میکند که ابتدا ویژگی ای که بیشترین همبستگی را با متغیر هدف دارد به لیست ویژگی های انتخاب شده که در ابتدا خالی است اضافه میکند سپس ویژگی ها را به ترتیب اندازه همبستگی با متغیر هدف به این صورت چک میکند که اگر با یکی از ویژگی ها در لیست ویژگی های انتخاب شده اندازه همبستگی بیشتر از حد استانه دارد آن را انتخاب نمیکند و در غیر این صورت آن ویژگی انتخاب میشود.



```

def select_features(X_train , columns , trashhold):
    """select features from train set based on correlation

Args:
    X_train ([dataframe]): [features for train]
    columns ([list]): [names of features sorted based on corr with target]
    trashhold ([float]): [trashhold for selecting features]

Returns:
    [list]: [names of selected features]
"""

corr = X_train.corr()
corr = corr.abs()
featuer_names = []
featuer_names.append(columns[0])
i = 2
for column in columns[1:]:
    flag = True
    for j in featuer_names:
        if corr[column].loc[j] > trashhold:
            flag = False
            break
    if flag:
        featuer_names.append(column)

return featuer_names

```

Listing : ۱۴

قسمت اصلی کد این بخش در صفحه بعدی امده است در این قسمت با استفاده از CV به ازای حد استانه های مختلف ویژگی ها انتخاب میشوند و الگوریتم رگرسیون خطی اجرا میشود. و میانگین score ها برای هر حد استانه محاسبه میشود همچنین نتایج خواسته شده در این قسمت به صورت زیر هست. مشاهده میشود بهترین حالت زمانی هست که هیچ ویژگی ای حذف نشود.

```

trashhold: 0.35 score: 0.5784039889748491
selected features: ['Temperature', 'Hour', 'Seasons', 'Functioning Day', 'Visibility (10m)', 'Snowfall (cm)', 'Rainfall(mm)', 'Wind speed (m/s)', 'Holiday']
trashhold: 0.4 score: 0.5234033691266868
selected features: ['Temperature', 'Hour', 'Solar Radiation (MJ/m2)', 'Seasons', 'Functioning Day', 'Visibility (10m)', 'Snowfall (cm)', 'Rainfall(mm)', 'Wind speed (m/s)', 'Holiday']
trashhold: 0.5 score: 0.5353429212018582
selected features: ['Temperature', 'Hour', 'Solar Radiation (MJ/m2)', 'Seasons', 'Functioning Day', 'Visibility (10m)', 'Snowfall (cm)', 'Rainfall(mm)', 'Wind speed (m/s)', 'Holiday']
trashhold: 0.6 score: 0.54469588878136
selected features: ['Temperature', 'Hour', 'Solar Radiation (MJ/m2)', 'Seasons', 'Functioning Day', 'Visibility (10m)', 'Humidity(%)', 'Snowfall (cm)', 'Rainfall(mm)', 'Wind speed (m/s)', 'Holiday']
trashhold: 0.7 score: 0.546695888780136
selected features: ['Temperature', 'Hour', 'Solar Radiation (MJ/m2)', 'Seasons', 'Functioning Day', 'Visibility (10m)', 'Humidity(%)', 'Snowfall (cm)', 'Rainfall(mm)', 'Wind speed (m/s)', 'Holiday']
trashhold: 0.8 score: 0.546695888780136
selected features: ['Temperature', 'Hour', 'Solar Radiation (MJ/m2)', 'Seasons', 'Functioning Day', 'Visibility (10m)', 'Humidity(%)', 'Snowfall (cm)', 'Rainfall(mm)', 'Wind speed (m/s)', 'Holiday']
trashhold: 1 score: 0.5476749593774566
selected features: ['Temperature', 'Hour', 'Dew point temperature', 'Solar Radiation (MJ/m2)', 'Seasons', 'Functioning Day', 'Visibility (10m)', 'Humidity(%)', 'Snowfall (cm)', 'Rainfall(mm)', 'Wind speed (m/s)', 'Holiday']

```

شکل ۱۷: نتایج خواسته شده



```
#initializing kfold validation
kf = KFold(n_splits = 5, shuffle = True, random_state = 2)
#this dict is for saving score for each trashhold
score = {}
#this dict is for saving selected features based on different trashholds
selected = {}
#trashholds for selecting features
trashhold = [.35 , .4 , .5 , .75 ,.9 , 1]
for i in trashhold:
    score[i] = 0
#performing kfold for different trashholds
for train_index, test_index in kf.split(features):
    #splitting test and train
    X_train, X_test = features.iloc[train_index], features.iloc[test_index]
    y_train, y_test = target[train_index], target[test_index]
    new_df = df.iloc[train_index]
    #sorting feature names based on corr with target
    corr_target = new_df.corr()[target_col]
    corr_target = corr_target.abs()
    columns = corr_target.sort_values(ascending = False)[1:].index

    for i in trashhold:
        #selecting features
        selected_features = select_features(X_train , columns , i)
        selected[i] = selected_features
        new_X_train = X_train[selected_features]
        new_X_test = X_test[selected_features]
        #performing regression
        regr = linear_model.LinearRegression()
        regr.fit(new_X_train, y_train)
        y_pred = regr.predict(new_X_test)
        score[i]+= regr.score(new_X_test,y_test)
#calculating maen of scores
for i in score.keys():
    score[i]/=5
```

Listing : ۱۵



۲.۴

پاراکتر الفا در رگرسیون لسو رگولاریزیشن را کنترل میکند. این پارامتر مانند گاما در رگولاریزیشن است با این تفاوت که در نرم یک ضریب ضرب میشود و با افزایش ان اندازه ضرایب کوچک تر میشود. کد مربوط به این بخش به صورت زیر است

```
#dict for saving scores for different alphas
score={}
alphas = [.9,.98 ,1 , 1.02 , 1.5]
#initializing kfold
kf = KFold(n_splits = 5, shuffle = True, random_state = 2)
for i in alphas:
    score[i] = 0
for train_index, test_index in kf.split(features):

    X_train, X_test = features.iloc[train_index], features.iloc[test_index]
    y_train, y_test = target[train_index], target[test_index]

    for alpha in alphas:
        clf = linear_model.Lasso(alpha=alpha)
        clf.fit(X_train , y_train)
        score[alpha]+= clf.score(X_test , y_test)

for i in score.keys():
    score[i]/=5

print('scores: ')
for i in score.keys():
    print('alpha: ' , i , 'score: ' , score[i])
#choosing best alpha
t=0
for i in score.keys():
    if score[i]>t:
        t = score[i]
        best_alpha = i
clf = linear_model.Lasso(alpha=best_alpha)
clf.fit(features , target)
print('dropped featuers with lasso: ',list(features.columns[clf.coef_==0]))
```

Listing : ۱۶

در کد بالا ما به ازای مقادیر مختلف الفا و با استفاده از CV الگوریتم رگرسیون لسو را اجرا میکنیم و بهترین الفا را انتخاب میکنیم. نتایج این بخش در صفحه بعد امده اند. مشاهده میشود که بهترین مقدار الفا برابر 0.9 میباشد.



```
scores:  
alpha: 0.9 score: 0.5443027269781562  
alpha: 0.98 score: 0.5438338755784449  
alpha: 1 score: 0.543709938845153  
alpha: 1.02 score: 0.5435830421267938  
alpha: 1.5 score: 0.5397821923357036  
dropped features with lasso: ['Dew point temperature', 'Snowfall (cm)']
```

شکل ۱۸: نتایج خواسته شده

۴.۴

در بخش دوم هیچ ویژگی حذف نشد اما در بخش سوم دو ویژگی Dew point temperature و snowfall حذف شدند. احتمال دارد که این ویژگی ها باعث بیش برآذش مدل شده باشند که در نتیجه لاسو ان ها را حذف میکند.

۵.۴

برای بهبود عملکرد مدل میتوانیم درجات بالاتر ویژگی ها را به مدل اضافه کنیم همچنین میتوانیم ترکیبات درجه بالاتر ویژگی ها را به مدل اضافه کنیم

۵

۱.۵

کد این بخش در صفحه بعد آمده است. در این کد ابتدا ردیف هایی که ۴ یا بیشتر مقدار گم شده داشتند از دیتابست حذف شدند که تعداد این ردیف ها ۴ عدد بود که با توجه به تعداد کم این ها احتمالاً حذف این ها تاثیر منفی ای روی نتیجه مدل ها نخواهد داشت. همچنین برای ویژگی های categorical مقادیر گم شده را با مد ویژگی جایگزین کردیم زیرا احتمال این که مقدار واقعی این مقادیر برابر با مد باشد بیشتر از سایر مقادیر است. همچنین برای ویژگی های عددی از میانگین ویژگی برای پر کردن مقادیر گم شده استفاده کردیم زیرا این کار میانگین ویژگی را تغییر نمیدهد.



```
#reading dataset
df = pd.read_csv('heart_failure_clinical_records_dataset.csv')
#replacing ? marks with nan values
df.replace({'?': np.nan}, inplace =True)
#selecting target column
target_name = 'DEATH_EVENT'
target = df[target_name]
#selecting features
features = df.drop(columns = [target_name])
features = features.astype('float64')
#selecting rows with null values
null_rows = features[features.isnull().sum(axis=1)!=0]

print('number of null values in rows with them:')
print(null_rows.isnull().sum(axis=1))

#dropping rows with 4 or more null values
new_features = features.drop([213,238,273,299])
target = target.drop([213,238,273,299])
#filling null values in categorical columns with mode
for column in ['sex' , 'diabetes' ]:
    new_features[column].fillna(int(new_features[column].mode()), inplace=True)
#filling null values in categorical columns with mean
for column in ['age' , 'creatinine_phosphokinase' , 'creatinine_phosphokinase','ejection_fraction','platelets','s':
    new_features[column].fillna(int(new_features[column].mean())), inplace=True)
#reset indices
new_features = new_features.reset_index(drop = True)
target = target.reset_index(drop=True)
#normalizing
for column in new_features.columns:
    new_features[column] = (new_features[column] - new_features[column].min())/(new_features[column].max() - new_features[column].min())

```

Listing :\V



۲.۵

کد این بخش به صورت زیر است

```
#selecting features
selector = SelectFromModel(estimator=LogisticRegression()).fit(new_features, target)
#sorting features based on thier importance in selector
x = [(new_features.columns[i] , selector.estimator_.coef_[0][i]) for i in range(len(new_features.columns))]
x.sort(key = lambda y:y[1] , reverse=True)

#calculating classifier score for first k imporant feature with CV
for k in range(12):
    clf = LogisticRegression()
    scores = cross_val_score(clf, new_features[[i[0] for i in x[:k+1]]], target, cv=5)
    print('score for ' , k+1 , 'features: ' , scores.mean())
```

Listing : ۱۸

در این کد ابتدا میزان مهم بودن هر ویژگی را بر اساس دسته بندی لاجستیک رگرسیون بدست می اوریم سپس ویژگی ها را بر اساس این مقادیر مرتب میکنیم و در یک حلقه در هر مرحله k ویژگی اول این لیست را انتخاب میکنیم و امتیاز مدل را محاسبه میکنیم . نتایج بدست امده به صورت زیر است. مشاهده میشود

```
score for 1 features:  0.6891525423728814
score for 2 features:  0.695819209039548
score for 3 features:  0.695819209039548
score for 4 features:  0.6890395480225988
score for 5 features:  0.695819209039548
score for 6 features:  0.6992655367231638
score for 7 features:  0.7026553672316384
score for 8 features:  0.6992090395480226
score for 9 features:  0.6958757062146892
score for 10 features: 0.695819209039548
score for 11 features: 0.7059322033898304
score for 12 features: 0.7738983050847458
```

شکل ۱۹: نتایج خواسته شده

بهترین حالت زمانی است که تمام ویژگی ها انتخاب شود



۲.۵

کد این قسمت به صورت زیر است

```
#initializing base classifiers
clf1 = LogisticRegression()
clf2 = RandomForestClassifier(n_estimators=50)
clf3 = GaussianNB()
clf4 = DecisionTreeClassifier(max_depth=4)
clf5 = KNeighborsClassifier(n_neighbors=7)
clf6 = SVC(gamma=.1, kernel='rbf' , probability=True)

#initializing voting classifiers
eclf1 = VotingClassifier(estimators=[('lr', clf1), ('rf', clf2),
                                      ('gnb', clf3)],
                         voting='soft')
eclf2 = VotingClassifier(estimators=[('lr', clf1), ('dt', clf4),
                                      ('knn', clf5)],
                         voting='soft')
eclf3 = VotingClassifier(estimators=[('svm', clf6), ('lr', clf1),
                                      ('dt', clf4)],
                         voting='soft')

for i , eclf in enumerate([eclf1 , eclf2 , eclf3]):
    scores = cross_val_score(eclf, new_features, target, cv=5)
    print('score for ' , i+1, 'st' , 'voting classifier: ' , scores.mean())
```

Listing : ۱۹

در این کد مدل رای گیری اول از سه دسته بند پایه لاجستیک رگرسیون و جنگل تصادفی و بیز ساده گاوی استفاده میکند . مدل رای گیری دوم از سه دسته بند پایه لاجستیک رگرسیون و درخت تصمیم و k نزدیک ترین همسایه استفاده میکند و مدل رای گیری سوم نیز از سه دسته بند پایه ماشین بردار پشتیبان و لاجستیک رگرسیون و درخت تصمیم استفاده میکند. نتایج این بخش به صورت زیر است. مشاهده میشود بهترین عملکرد

```
score for 1 st voting classifier:  0.7434463276836157
score for 2 st voting classifier:  0.7102824858757062
score for 3 st voting classifier:  0.713728813559322
```

شکل ۲۰: نتایج خواسته شده

مربوط به مدل رای گیر اول است.



۴.۵

کد این بخش به صورت زیر است

```
#initializing classifiers
clf1 = LogisticRegression()
clf2 = RandomForestClassifier(n_estimators=50)
clf3 = GaussianNB()

for i , clf in [('lr' , clf1) , ('rf',clf2) , ('gnb' ,clf3)]:
    scores = cross_val_score(clf, new_features, target, cv=5)
    print('score for ' , i , 'classifier: ' , scores.mean())
```

Listing : ۲۰

در بخش قبل دسته بند های پایه مربوط به بهترین مدل عبارت بودند از لاجستیک رگرسیون و جنگل تصادفی و بیز ساده گاوی. در این بخش نیز برای این دسته بندی کننده ها امتیاز مدل را بوسیله CV بدست اورده ایم که نتایج به صورت زیر است . مشاهده میشود که لاجستیک رگرسیون عملکرد بهتری نسبت به مدل

```
score for lr classifier:  0.7738983050847458
score for rf classifier:  0.7036723163841808
score for gnb classifier:  0.7601129943502825
```

شکل ۲۱: نتایج خواسته شده

رای گیری دارد که یکی از دلایل این میتواند باشد که برای انتخاب ویژگی ها از این دسته بندی کننده استفاده کردیم. اما دو دسته بندی دیگر عملکرد ضعیف تری نسبت به مدل رای گیری دارند. و میتوان گفت ترکیب ان ها قوی تراز تک تک ان هاست.