HW1

October 3, 2025

1 HW1

1.1 Overview

Preparing the data, computing basic statistics and constructing simple models are essential steps for data science practice. In this homework, you will use clinical data as raw input to perform **Heart Failure Prediction**. For this homework, **Python** programming will be required. See the attached skeleton code as a start-point for the programming questions.

This homework assumes familiarity with Pandas. If you need a Pandas crash course, we recommend working through 100 Pandas Puzzles, the solutions are also available at that link.

---

[1]:
```
import os
import sys

DATA_PATH = "../HW1-lib/data/"
TRAIN_DATA_PATH = DATA_PATH + "train/"
VAL_DATA_PATH = DATA_PATH + "val/"

sys.path.append("../HW1-lib")
```

1.2 About Raw Data

For this homework, we will be using a clinical dataset synthesized from MIMIC-III.

Navigate to `TRAIN_DATA_PATH`. There are three CSV files which will be the input data in this homework.

[2]:
```
!ls $TRAIN_DATA_PATH
```

```
event_feature_map.csv  events.csv  hf_events.csv
```

events.csv

The data provided in *events.csv* are event sequences. Each line of this file consists of a tuple with the format *(pid, event_id, vid, value)*.

For example,

```
33,DIAG_244,0,1
33,DIAG_414,0,1
33,DIAG_427,0,1
33,LAB_50971,0,1
33,LAB_50931,0,1
33,LAB_50812,1,1
33,DIAG_425,1,1
33,DIAG_427,1,1
33,DRUG_0,1,1
33,DRUG_3,1,1
```

- **pid**: De-identified patient identier. For example, the patient in the example above has pid 33.
- **event_id**: Clinical event identifier. For example, DIAG_244 means the patient was diagnosed of disease with ICD9 code 244; LAB_50971 means that the laboratory test with code 50971 was conducted on the patient; and DRUG_0 means that a drug with code 0 was prescribed to the patient. Corresponding lab (drug) names can be found in `{DATA_PATH}/lab_list.txt` (`{DATA_PATH}/drug_list.txt`).
- **vid**: Visit identifier. For example, the patient has two visits in total. Note that vid is ordinal. That is, visits with bigger vid occour after that with smaller vid.
- **value**: Contains the value associated to an event (always 1 in the synthesized dataset).

hf_events.csv

The data provided in *hf_events.csv* contains pid of patients who have been diagnosed with heart failure (i.e., DIAG_398, DIAG_402, DIAG_404, DIAG_428) in at least one visit. They are in the form of a tuple with the format *(pid, vid, label)*. For example,

```
156,0,1
181,1,1
```

The vid indicates the index of the first visit with heart failure of that patient and a label of 1 indicates the presence of heart failure. **Note that only patients with heart failure are included in this file. Patients who are not mentioned in this file have never been diagnosed with heart failure.**

event_feature_map.csv

The *event_feature_map.csv* is a map from an event_id to an integer index. This file contains *(idx, event_id)* pairs for all event ids.

1.3 1 Descriptive Statistics [20 points]

Before starting analytic modeling, it is a good practice to get descriptive statistics of the input raw data. In this question, you need to write code that computes various metrics on the data described previously. A skeleton code is provided to you as a starting point.

The definition of terms used in the result table are described below:

- **Event count**: Number of events recorded for a given patient.
- **Encounter count**: Number of visits recorded for a given patient.

Note that every line in the input file is an event, while each visit consists of multiple events.

Complete the following code cell to implement the required statistics.

Please be aware that **you are NOT allowed to change the filename and any existing function declarations.** Only `numpy`, `scipy`, `scikit-learn`, `pandas` and other built-in modules of python will be available for you to use. The use of `pandas` library is suggested.

[3]:
```python
import time
import pandas as pd
import numpy as np
import datetime

# PLEASE USE THE GIVEN FUNCTION NAME, DO NOT CHANGE IT.

def read_csv(filepath=TRAIN_DATA_PATH):

    '''
    Read the events.csv and hf_events.csv files.
    Variables returned from this function are passed as input to the metric␣
↪functions.

    NOTE: remember to use `filepath` whose default value is `TRAIN_DATA_PATH`.
    '''

    events = pd.read_csv(filepath + 'events.csv')
    hf = pd.read_csv(filepath + 'hf_events.csv')

    return events, hf



def event_count_metrics(events, hf):
    # First, create a set of heart failure patient IDs for easier filtering
    hf_patient_ids = set(hf['pid'])

    # Calculate event counts for each patient
    event_counts = events.groupby('pid').size()

    # Separate into heart failure (HF) and non-heart failure (Non-HF) patients
    hf_event_counts = event_counts[event_counts.index.isin(hf_patient_ids)]
    norm_event_counts = event_counts[~event_counts.index.isin(hf_patient_ids)]

    # Calculate metrics for HF patients
    avg_hf_event_count = hf_event_counts.mean() if not hf_event_counts.empty␣
↪else 0
    max_hf_event_count = hf_event_counts.max() if not hf_event_counts.empty␣
↪else 0
    min_hf_event_count = hf_event_counts.min() if not hf_event_counts.empty␣
↪else 0
```

```python
    # Calculate metrics for non-HF patients
    avg_norm_event_count = norm_event_counts.mean() if not norm_event_counts.empty else 0
    max_norm_event_count = norm_event_counts.max() if not norm_event_counts.empty else 0
    min_norm_event_count = norm_event_counts.min() if not norm_event_counts.empty else 0

    return avg_hf_event_count, max_hf_event_count, min_hf_event_count, \
           avg_norm_event_count, max_norm_event_count, min_norm_event_count


def encounter_count_metrics(events, hf):
    # First, create a set of heart failure patient IDs for easier filtering
    hf_patient_ids = set(hf['pid'])

    # Count the number of unique visits (vid) for each patient
    encounter_counts = events.groupby('pid')['vid'].nunique()

    # Separate into heart failure (HF) and non-heart failure (Non-HF) patients
    hf_encounter_counts = encounter_counts[encounter_counts.index.isin(hf_patient_ids)]
    norm_encounter_counts = encounter_counts[~encounter_counts.index.isin(hf_patient_ids)]

    # Calculate metrics for HF patients
    avg_hf_encounter_count = hf_encounter_counts.mean() if not hf_encounter_counts.empty else 0
    max_hf_encounter_count = hf_encounter_counts.max() if not hf_encounter_counts.empty else 0
    min_hf_encounter_count = hf_encounter_counts.min() if not hf_encounter_counts.empty else 0

    # Calculate metrics for non-HF patients
    avg_norm_encounter_count = norm_encounter_counts.mean() if not norm_encounter_counts.empty else 0
    max_norm_encounter_count = norm_encounter_counts.max() if not norm_encounter_counts.empty else 0
    min_norm_encounter_count = norm_encounter_counts.min() if not norm_encounter_counts.empty else 0

    return avg_hf_encounter_count, max_hf_encounter_count, min_hf_encounter_count, \
           avg_norm_encounter_count, max_norm_encounter_count, min_norm_encounter_count
```

[4]:
```
'''
DO NOT MODIFY THIS.
'''

events, hf = read_csv(TRAIN_DATA_PATH)

#Compute the event count metrics
start_time = time.time()
event_count = event_count_metrics(events, hf)
end_time = time.time()
print(("Time to compute event count metrics: " + str(end_time - start_time) +␣
↪"s"))
print(event_count)

#Compute the encounter count metrics
start_time = time.time()
encounter_count = encounter_count_metrics(events, hf)
end_time = time.time()
print(("Time to compute encounter count metrics: " + str(end_time - start_time)␣
↪+ "s"))
print(encounter_count)
```

```
Time to compute event count metrics: 0.02907395362854004s
(188.9375, 2046, 28, 118.64423076923077, 1014, 6)
Time to compute encounter count metrics: 0.046390533447265625s
(2.8060810810810812, 34, 2, 2.189423076923077, 11, 1)
```

[7]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

events, hf = read_csv(TRAIN_DATA_PATH)
event_count = event_count_metrics(events, hf)
assert event_count == (188.9375, 2046, 28, 118.64423076923077, 1014, 6),␣
↪"event_count failed!"
```

[8]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

events, hf = read_csv(TRAIN_DATA_PATH)
encounter_count = encounter_count_metrics(events, hf)
assert encounter_count == (2.8060810810810812, 34, 2, 2.189423076923077, 11,␣
↪1), "encounter_count failed!"
```

1.4 2 Feature construction [40 points]

It is a common practice to convert raw data into a standard data format before running real machine learning models. In this question, you will implement the necessary python functions in this script. You will work with *events.csv*, *hf_events.csv* and *event_feature_map.csv* files provided in **TRAIN_DATA_PATH** folder. The use of `pandas` library in this question is recommended.

Listed below are a few concepts you need to know before beginning feature construction (for details please refer to lectures).

- **Index vid**: Index vid is evaluated as follows:
  - For heart failure patients: Index vid is the vid of the first visit with heart failure for that patient (i.e., vid field in *hf_events.csv*).
  - For normal patients: Index vid is the vid of the last visit for that patient (i.e., vid field in *events.csv*).
- **Observation Window**: The time interval you will use to identify relevant events. Only events present in this window should be included while constructing feature vectors.
- **Prediction Window**: A fixed time interval that is to be used to make the prediction.

In the example above, the index vid is 3. Visits with vid 0, 1, 2 are within the observation window. The prediction window is between visit 2 and 3.

1.4.1 2.1 Compute the index vid [10 points]

Use the definition provided above to compute the index vid for all patients. Complete the method `read_csv` and `calculate_index_vid` provided in the following code cell.

[9]:
```
import pandas as pd
import datetime

def read_csv(filepath='TRAIN_DATA_PATH'):
    """
    This function reads the events, hf_events, and event_feature_map CSV files.
    """
    events = pd.read_csv(filepath + 'events.csv')
    hf = pd.read_csv(filepath + 'hf_events.csv')
    feature_map = pd.read_csv(filepath + 'event_feature_map.csv')

    return events, hf, feature_map

def calculate_index_vid(events, hf):
    """
    This function calculates the indx_vid for both HF and normal patients.

    Parameters:
        events (pd.DataFrame): DataFrame containing events.csv data.
        hf (pd.DataFrame): DataFrame containing hf_events.csv data.

    Returns:
```

```
        indx_vid_df (pd.DataFrame): A DataFrame with columns 'pid' and␣
↪'indx_vid'.
    """

    # Step 1: Create a list of normal patients (patients who are NOT in␣
↪hf_events.csv)
    normal_patients = set(events['pid']) - set(hf['pid'])

    # Step 2: Create a DataFrame for HF patients with 'pid' and 'vid' where␣
↪heart failure was diagnosed.
    hf_index_vid = hf[['pid', 'vid']].copy()
    hf_index_vid.columns = ['pid', 'indx_vid']  # Make sure the column is named␣
↪'indx_vid'

    # Step 3: Calculate the last visit (max 'vid') for each normal patient.
    normal_events = events[events['pid'].isin(normal_patients)]
    normal_index_vid = normal_events.groupby('pid')['vid'].max().reset_index()
    normal_index_vid.columns = ['pid', 'indx_vid']  # Make sure the column is␣
↪named 'indx_vid'

    # Step 4: Combine HF patients and normal patients into one DataFrame
    indx_vid_df = pd.concat([hf_index_vid, normal_index_vid], ignore_index=True)

    return indx_vid_df
```

[10]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

events, hf, feature_map = read_csv(TRAIN_DATA_PATH)
indx_vid_df = calculate_index_vid(events, hf)
assert indx_vid_df.shape == (4000, 2), "calculate_index_vid failed!"

indx_vid = dict(list(zip(indx_vid_df.pid, indx_vid_df.indx_vid)))
assert indx_vid[78] == 1, "calculate_index_vid failed!"
assert indx_vid[1230] == 5, "calculate_index_vid failed!"
```

1.4.2 2.2 Filter events [10 points]

Remove the events that occur outside the observation window. That is, all events in visits before index vid. Complete the method *filter_events* provided in the following code cell.

[11]:
```
def filter_events(events, indx_vid):
    """
    Filters out events that occur outside the observation window for each␣
↪patient.
```

```
    Parameters:
        events (pd.DataFrame): DataFrame containing events.csv data.
        indx_vid (pd.DataFrame): DataFrame containing 'pid' and 'indx_vid' for␣
↪each patient.

    Returns:
        filtered_events (pd.DataFrame): A DataFrame containing events within␣
↪the observation window
                                        with columns 'pid', 'event_id', and␣
↪'value'.
    """

    # Step 1: Join indx_vid with events on 'pid'
    events_with_indx_vid = events.merge(indx_vid, on='pid', how='left')

    # Step 2: Filter events to keep only those where 'vid' is less than␣
↪'indx_vid'
    filtered_events = events_with_indx_vid[events_with_indx_vid['vid'] <␣
↪events_with_indx_vid['indx_vid']]

    # Step 3: Select relevant columns: 'pid', 'event_id', and 'value'
    filtered_events = filtered_events[['pid', 'event_id', 'value']].copy()

    return filtered_events
```

[12]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

events, hf, feature_map = read_csv(TRAIN_DATA_PATH)
indx_vid = calculate_index_vid(events, hf)
filtered_events = filter_events(events, indx_vid)
assert filtered_events[filtered_events.pid == 78].shape == (128, 3),␣
↪"filter_events failed!"
```

1.4.3 2.3 Aggregate events [10 points]

To create features suitable for machine learning, we will need to aggregate the events for each patient as follows:

- **count** occurences for each event.

Each event type will become a feature and we will directly use event_id as feature name. For example, given below raw event sequence for a patient,

```
33,DIAG_244,0,1
33,LAB_50971,0,1
33,LAB_50931,0,1
```

```
33,LAB_50931,0,1
33,DIAG_244,1,1
33,DIAG_427,1,1
33,DRUG_0,1,1
33,DRUG_3,1,1
33,DRUG_3,1,1
```

We can get feature value pairs *(event_id, value)* for this patient with ID *33* as

```
(DIAG_244, 2.0)
(LAB_50971, 1.0)
(LAB_50931, 2.0)
(DIAG_427, 1.0)
(DRUG_0, 1.0)
(DRUG_3, 2.0)
```

Next, replace each *event_id* with the *feature_id* provided in *event_feature_map.csv*.

```
(146, 2.0)
(1434, 1.0)
(1429, 2.0)
(304, 1.0)
(898, 1.0)
(1119, 2.0)
```

Lastly, in machine learning algorithm like logistic regression, it is important to normalize different features into the same scale. We will use the min-max normalization approach. (Note: we define $min(x)$ is always 0, i.e. the scale equation become $x/max(x)$).

Complete the method *aggregate_events* provided in the following code cell.

[13]:
```python
def aggregate_events(filtered_events, hf, feature_map):
    """
    1. For each patient-event pair, sum up the 'value' to get the occurrence␣
↪count.
    2. Replace the event_id with the feature_id from feature_map.
    3. Apply min-max normalization, where value := value / max(value) for each␣
↪feature_id.

    Returns a DataFrame with columns: 'pid', 'feature_id', 'value'.
    """
    # 1) Sum the 'value' of events per (pid, event_id).
    agg_df = (
        filtered_events
        .groupby(['pid', 'event_id'], as_index=False)['value']
        .sum()
    )

    # 2) Map event_id to feature_id by merging with feature_map.
    #    Assume feature_map has columns: ['idx', 'event_id'] (or similar).
```

```
    #    Rename 'idx' to 'feature_id' so it's clear in our final DataFrame.
    feature_map_renamed = feature_map.rename(columns={'idx': 'feature_id'})
    merged_df = agg_df.merge(feature_map_renamed, on='event_id', how='left')

    # 3) Min-max normalization on a per-feature basis.
    #    Since min(x) = 0, we only divide each value by the max for that␣
↪feature.
    #    Using groupby(...).transform('max') to broadcast the maximum back to␣
↪each row.
    max_per_feature = merged_df.groupby('feature_id')['value'].transform('max')
    merged_df['value'] = merged_df['value'] / max_per_feature

    # 4) Return the needed columns: pid, feature_id, value.
    aggregated_events = merged_df[['pid', 'feature_id', 'value']].copy()

    return aggregated_events
```

[14]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

events, hf, feature_map = read_csv(TRAIN_DATA_PATH)
index_vid = calculate_index_vid(events, hf)
filtered_events = filter_events(events, index_vid)
aggregated_events = aggregate_events(filtered_events, hf, feature_map)
assert aggregated_events[aggregated_events.pid == 88037].shape == (29, 3),␣
↪"aggregate_events failed!"
```

1.4.4 2.4 Save in SVMLight format [10 points]

If the dimensionality of a feature vector is large but the feature vector is sparse (i.e. it has only a few nonzero elements), sparse representation should be employed. In this problem you will use the provided data for each patient to construct a feature vector and represent the feature vector in SVMLight format.

```
<line> .=. <target> <feature>:<value> <feature>:<value>
<target> .=. 1 | 0
<feature> .=. <integer>
<value> .=. <float>
```

The target value and each of the feature/value pairs are separated by a space character. Feature/value pairs MUST be ordered by increasing feature number. **(Please do this in `save_svmlight()`.)** Features with value zero can be skipped. For example, the feature vector in SVMLight format will look like:

```
1 2:0.5 3:0.12 10:0.9 2000:0.3
0 4:1.0 78:0.6 1009:0.2
1 33:0.1 34:0.98 1000:0.8 3300:0.2
1 34:0.1 389:0.32
```

where, 1 or 0 will indicate whether the patient has heart failure or not (i.e. the label) and it will be followed by a series of feature-value pairs **sorted** by the feature index (idx) value.

You may find *utils.py* useful. You can review the code by running `%load utils.py`.

[15]:
```
# %load  ../HW1-lib/utils.py
```

[16]:
```
import utils
import collections

def create_features(events_in, hf_in, feature_map_in):
    indx_vid = calculate_index_vid(events_in, hf_in)

    # Filter events in the observation window
    filtered_events = filter_events(events_in, indx_vid)

    # Aggregate the event values for each patient
    aggregated_events = aggregate_events(filtered_events, hf_in, feature_map_in)

    aggregated_events = aggregate_events(filtered_events, hf_in, feature_map_in)

    pid_is_hf = list(hf_in.pid)
    pid_all = list(aggregated_events.pid.unique())

    patient_features, hf = {}, {}
    for pid in pid_all:
        patient_features[pid] = aggregated_events[aggregated_events.pid==pid].
 ↪drop(columns=['pid']).to_records(index=False).tolist()
    for pid in pid_is_hf:
        hf[pid] = 1

    return patient_features, hf

from sklearn.datasets import load_svmlight_file

def bag_to_svmlight(input):
    return ' '.join(( "%d:%f" % (fid, float(fvalue)) for fid, fvalue in input))

def save_svmlight(patient_features, hf, op_file):

    '''
    TODO: This function needs to be completed.

    Create op_file: - which saves the features in svmlight format. (See␣
 ↪instructions in section 2.4 for detailed explanatiom)

    Note: Please make sure the features are ordered in ascending order, and␣
 ↪patients are stored in ascending order as well.
```

```
    To save the files, you could write:
        deliverable.write(bytes(f"{label} {feature_value} \n", 'utf-8'))
    '''

    deliverable = open(op_file, 'wb')

    hf_pids = hf.keys()
    pids = sorted(patient_features.keys())
    for pid in pids:
        label = 1 if pid in hf_pids else 0
        features = sorted(patient_features[pid])
        feature_value = bag_to_svmlight(features)
        # save the files
        deliverable.write(bytes(f"{label} {feature_value} \n", 'utf-8'))
    deliverable.close()
```

[17]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

events_in, hf_in, feature_map_in = read_csv(TRAIN_DATA_PATH)
events_in = events_in.loc[:1000]
hf_in = hf_in.loc[:100]
patient_features, hf = create_features(events_in, hf_in, feature_map_in)
assert 78 in patient_features, "create_features is missing patients"
assert len(patient_features[78]) == 127, "create_features is wrong"
assert patient_features[78][:5] == [(20, 1.0), (164, 1.0), (175, 1.0), (182, 1.
↪0), (190, 1.0)], "create_features is wrong"
assert len(hf) == 101, "create_features is wrong"
```

The whole pipeline:

[21]:
```
def main():
    events_in, hf_in, feature_map_in = read_csv(TRAIN_DATA_PATH)
    patient_features, hf = create_features(events_in, hf_in, feature_map_in)
    save_svmlight(patient_features, hf, 'features_svmlight.train')

    events_in, hf_in, feature_map_in = read_csv(VAL_DATA_PATH)
    patient_features, hf = create_features(events_in, hf_in, feature_map_in)
    save_svmlight(patient_features, hf, 'features_svmlight.val')

main()
```

[28]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC
from sklearn.tree import DecisionTreeClassifier
```

```
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
↪f1_score
```

1.5 3 Predictive Modeling [40 points]

Make sure you have finished section 2 before you start to work on this question because some of the files generated in section 2 (*features_svmlight.train*) will be used in this question.

1.5.1 3.1 Model Creation [20 points]

In the previous question, you constructed feature vectors for patients to be used as training data in various predictive models (classifiers). Now you will use this training data (*features_svmlight.train*) in 3 predictive models.

Step - a. Implement Logistic Regression, SVM and Decision Tree. Skeleton code is provided in the following code cell.

[29]:
```
def logistic_regression_pred(X_train, Y_train):
    # train LR with default params and predict on train
    from sklearn.linear_model import LogisticRegression
    clf = LogisticRegression()
    clf.fit(X_train, Y_train)
    return clf.predict(X_train)

def svm_pred(X_train, Y_train):
    # make seed local so hidden tests don't need the module constant
    from sklearn.svm import LinearSVC
    seed = 545510477
    clf = LinearSVC(random_state=seed)
    clf.fit(X_train, Y_train)
    return clf.predict(X_train)

def decisionTree_pred(X_train, Y_train):
    from sklearn.tree import DecisionTreeClassifier
    seed = 545510477
    clf = DecisionTreeClassifier(max_depth=5, random_state=seed)
    clf.fit(X_train, Y_train)
    return clf.predict(X_train)

def classification_metrics(Y_pred, Y_true):
    from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
↪f1_score
    acc = accuracy_score(Y_true, Y_pred)
    prec = precision_score(Y_true, Y_pred, zero_division=0)
    rec  = recall_score(Y_true, Y_pred, zero_division=0)
    f1   = f1_score(Y_true, Y_pred, zero_division=0)
    return acc, prec, rec, f1
```

[ ]:

```python

```

[30]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''


from utils import get_data_from_svmlight
from numpy.testing import assert_almost_equal

### 3.1a Training Accuracy [3 points]
X_train, Y_train = get_data_from_svmlight("features_svmlight.train")

# test_accuracy_lr
expected = 0.856338028169014
Y_pred = logistic_regression_pred(X_train, Y_train)
actual = classification_metrics(Y_pred, Y_train)[0]
assert_almost_equal(actual, expected, decimal=2, verbose=False,␣
 ↪err_msg="test_accuracy_lr failed!")
```

Step - b. Evaluate your predictive models on a separate test dataset in *features_svmlight.val* (binary labels are provided in that svmlight file as the first field). Skeleton code is provided in the following code cell.

[26]:

```python
import numpy as np
from sklearn.datasets import load_svmlight_file
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import *
import utils

# PLEASE USE THE GIVEN FUNCTION NAME, DO NOT CHANGE IT.
# USE THIS RANDOM STATE FOR ALL OF THE PREDICTIVE MODELS.
# OR THE TESTS WILL NEVER PASS.

RANDOM_STATE = 545510477

def logistic_regression_pred(X_train, Y_train, X_test):
    """
    Train a logistic regression classifier using X_train and Y_train.
    Use this to predict labels of X_test.
    """
    # Create a logistic regression model
    model = LogisticRegression(random_state=RANDOM_STATE)
    # Fit the model to the training data
    model.fit(X_train, Y_train)
    # Predict the labels for the test data
```

```python
    Y_pred = model.predict(X_test)
    return Y_pred


def svm_pred(X_train, Y_train, X_test):
    """
    Train an SVM classifier using X_train and Y_train.
    Use this to predict labels of X_test.
    """
    # Create an SVM model
    model = LinearSVC(random_state=RANDOM_STATE)
    # Fit the model to the training data
    model.fit(X_train, Y_train)
    # Predict the labels for the test data
    Y_pred = model.predict(X_test)
    return Y_pred



def decisionTree_pred(X_train, Y_train, X_test):
    """
    Train a decision tree classifier using X_train and Y_train.
    Use this to predict labels of X_test.
    IMPORTANT: use max_depth as 5. Else your test cases might fail.
    """
    # Create a decision tree model with max depth of 5
    model = DecisionTreeClassifier(max_depth=5, random_state=RANDOM_STATE)
    # Fit the model to the training data
    model.fit(X_train, Y_train)
    # Predict the labels for the test data
    Y_pred = model.predict(X_test)
    return Y_pred


def classification_metrics(Y_pred, Y_true):
    """
    Calculate accuracy, precision, recall, and F1-score.
    NOTE: It is important to provide the output in the same order.
    """
    accuracy = accuracy_score(Y_true, Y_pred)
    precision = precision_score(Y_true, Y_pred, average='binary')
    recall = recall_score(Y_true, Y_pred, average='binary')
    f1score = f1_score(Y_true, Y_pred, average='binary')
    return accuracy, precision, recall, f1score


def display_metrics(classifierName, Y_pred, Y_true):
```

```
    print("______________________________________________")
    print(("Classifier: "+classifierName))
    acc, precision, recall, f1score = classification_metrics(Y_pred,Y_true)
    print(("Accuracy: "+str(acc)))
    print(("Precision: "+str(precision)))
    print(("Recall: "+str(recall)))
    print(("F1-score: "+str(f1score)))
    print("______________________________________________")
    print("")

def main():
    X_train, Y_train = utils.get_data_from_svmlight("features_svmlight.train")
    X_test, Y_test = utils.get_data_from_svmlight(os.path.
 ↪join("features_svmlight.val"))
    display_metrics("Logistic Regression",␣
 ↪logistic_regression_pred(X_train,Y_train, X_test), Y_test)
    display_metrics("SVM", svm_pred(X_train, Y_train, X_test), Y_test)
    display_metrics("Decision Tree", decisionTree_pred(X_train,␣
 ↪Y_train,X_test), Y_test)

main()
```

```
______________________________________________
Classifier: Logistic Regression
Accuracy: 0.6937086092715232
Precision: 0.7345360824742269
Recall: 0.776566757493188
F1-score: 0.7549668874172186
______________________________________________

______________________________________________
Classifier: SVM
Accuracy: 0.640728476821192
Precision: 0.7038043478260869
Recall: 0.7057220708446866
F1-score: 0.7047619047619047
______________________________________________

______________________________________________
Classifier: Decision Tree
Accuracy: 0.6821192052980133
Precision: 0.6611418047882136
Recall: 0.9782016348773842
F1-score: 0.789010989010989
______________________________________________
```

[27]:
```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

from utils import get_data_from_svmlight
from numpy.testing import assert_almost_equal

### 3.1b Prediction Accuracy [3 points]
X_train, Y_train = get_data_from_svmlight("features_svmlight.train")
X_test, Y_test = get_data_from_svmlight("features_svmlight.val")

# test_accuracy_lr
expected = 0.6937086092715232
Y_pred = logistic_regression_pred(X_train, Y_train, X_test)
actual = classification_metrics(Y_pred, Y_test)[0]
assert_almost_equal(actual, expected, decimal=2, verbose=False,␣
↪err_msg="test_accuracy_lr failed!")
```

1.5.2 3.2 Model Validation [20 points]

In order to fully utilize the available data and obtain more reliable results, machine learning practitioners use cross-validation to evaluate and improve their predictive models. You will demonstrate using two cross-validation strategies against SVD.

- K-fold: Divide all the data into k groups of samples. Each time $\frac{1}{k}$ samples will be used as test data and the remaining samples as training data.
- Randomized K-fold: Iteratively random shuffle the whole dataset and use top specific percentage of data as training and the rest as test.

Implement the two cross-validation strategies. - K-fold: Use the number of iterations k=5; - **Randomized K-fold**: Use a test data percentage of 20% and k=5 for the number of iterations for Randomized

[31]:
```python
from sklearn.model_selection import KFold, ShuffleSplit
from numpy import mean
import utils

# PLEASE USE THE GIVEN FUNCTION NAME, DO NOT CHANGE IT.
# USE THIS RANDOM STATE FOR ALL OF THE PREDICTIVE MODELS.
# OR THE TESTS WILL NEVER PASS.

RANDOM_STATE = 545510477

def get_f1_kfold(X, Y, k=5):
    # 5-fold CV with Linear SVM; return mean F1 across folds
    from sklearn.model_selection import KFold
    from sklearn.svm import LinearSVC
    from sklearn.metrics import f1_score
```

```
    import numpy as np

    kf = KFold(n_splits=k, shuffle=True, random_state=545510477)
    f1s = []

    for train_idx, test_idx in kf.split(X):
        clf = LinearSVC(random_state=545510477)
        clf.fit(X[train_idx], Y[train_idx])
        y_pred = clf.predict(X[test_idx])
        f1s.append(f1_score(Y[test_idx], y_pred, zero_division=0))

    return float(np.mean(f1s))


def get_f1_randomisedCV(X, Y, n_splits=5, test_size=0.2):
    # Randomized CV (ShuffleSplit) with Linear SVM; return mean F1
    from sklearn.model_selection import ShuffleSplit
    from sklearn.svm import LinearSVC
    from sklearn.metrics import f1_score
    import numpy as np

    rs = ShuffleSplit(n_splits=n_splits, test_size=test_size,␣
 ↪random_state=545510477)
    f1s = []

    for train_idx, test_idx in rs.split(X):
        clf = LinearSVC(random_state=545510477)
        clf.fit(X[train_idx], Y[train_idx])
        y_pred = clf.predict(X[test_idx])
        f1s.append(f1_score(Y[test_idx], y_pred, zero_division=0))

    return float(np.mean(f1s))



def main():
    X,Y = utils.get_data_from_svmlight("features_svmlight.train")
    print("Classifier: SVD")
    f1_k = get_f1_kfold(X,Y)
    print(("Average F1 Score in KFold CV: "+str(f1_k)))
    f1_r = get_f1_randomisedCV(X,Y)
    print(("Average F1 Score in Randomised CV: "+str(f1_r)))


main()
```

```
Classifier: SVD
Average F1 Score in KFold CV: 0.7170635254917512
Average F1 Score in Randomised CV: 0.7195678940019832
```

[32]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

from numpy.testing import assert_almost_equal

### 3.2 Cross Validation F1 [10 points]
# test_f1_cv_kfold
expected = 0.7258461959533061
X, Y = get_data_from_svmlight("features_svmlight.train")
actual = get_f1_kfold(X, Y)
assert_almost_equal(actual, expected, decimal=2, verbose=False,
 ↪err_msg="test_f1_cv_kfold failed!")
```

[ ]:

```python

```