INFO-I526 APPLIED MACHINE LEARNING – SPRING 2017
Carlos Sathler (cssathler@gmail.com)

Programming Assignment #1

1. Use adult.data as your training set and adult.test as your test set.
   We used the following data for this project:
   Training dataset: adult.data
   Test dataset: adult.test
   Both files are available from http://archive.ics.uci.edu/ml/datasets/Adult

2. You may want to consider creating a validation set from the training set to help you select hyper-parameters. Hyper-parameters you may want to tune are:
   a) the threshold value for Logistic Regression/Naïve Bayes;
   b) the depth of the decision tree;
   c) the value of k in Nearest Neighbors.

   The following steps describe the tuning and training process we used for each algorithm:
   1. Training dataset (adult.data) was divided into train' and test' partitions (80/20% split)
   2. Train' partition was processed using kfold (k=10) to tune hyper parameter for algorithm.
   3. Each parameter value was tested for 10 kfold configurations using the corresponding validation dataset to compute score; a mean score was obtained for each parameter value.
   4. The best parameter value was thus selected and used against the test' dataset for final reporting algorithm score against the training dataset (please see program output in Appendix A and also in Canvas file "predict_adult.out").
   5. Trained function against the training dataset (adult.data).

   The following table lists hyper parameters tuned for each algorithm:

Algorithm	Parameter
Naïve Bayes	None known in sklearn
Decision Tree	Tree Depth
KNN	K neighbors
Logistic Regression	Penalty (l1, l2) and C

3. Take a look at the data and get a feel for the types of feature values you observe.
   We include in Appendix B a series of plots generated from Weka from a load of test.data in csv format into Weka. Please refer to Appendix B.

4. Preprocess the data. Make sure to explain how you preprocessed the data and why.
   Data was preprocessed after load with a general pre-processing routine. After general pre-processing, the data was further pre-processed with specific actions prior to submission to each of the learning algorithms.
   General preprocessing consisted of:
   - Removed rows with missing values in columns "workclass", "occupation" and "native_country"
   - Dropped column "education" since it's redundant (it matches column "education-num")
   - Converted all categorical columns to numbers
   - Converted all columns to floats

   Pre-processing specific to each algorithm is described under question b)

   a) It is important to note that some of the features are missing values (indicated with a "?"). How will you handle missing feature values? Explain your solution and why you chose it. You can treat a missing feature value as its own feature value, but this might not be the best solution, so you should occupation and native_country. These are all categorical features. Even though we could **have** used mode as a measure of centrality to impute missing values we opted to drop rows containing missing data to avoid adding bias to the training dataset.

b) Some of the feature values are continuous and some are categorical. Do you want to discretize the continuous features? If so, why? Do you want to discretize the feature values for all of the classifiers? It is okay to preprocess the data differently for each of the classifiers, just make sure to explain what you did and why.

The following table summarizes additional pre-processing for each specific algorithm:

Algorithm	Pre-Processing (What)	Reasoning (Why)
Naïve Bayes	Discretized continuous features: age, fnlwgt, capital_gain, capital_loss, hours_per_week Replaced original feature values with upper bound of percentile for the feature range.	Initially thought discretization of continuous variables was required since P(X=x|Y=y) = 0 if x is a number in the Real set. Found that not to be the case as sklearn performs better for Naïve Bayes with continuous feature values.
Decision Tree	Tried to discretize continuous features using the same pre-processing function initially used for Naïve Bays. In the end used original continuous values since sklearn Decision Tree Classifier algorithm performed better that way.	Discretized continuous features under the assumption that calculation of entropy would require discrete values. However, sklearn algorithm performed better with continuous feature values.
KNN	Same as Naïve Bayes. Also tried using continuous features, which yielded poorer results.	Discretized continuous features to increase feature grouping hoping to improve clustering of data points.
Logistic Regression	Standardized continuous features and converted categorical features into dummy variables.	Since logistic regression learns a liner model we must prepare features accordingly. Standardization will give equal weight to features like age (0-100 range) and capital_gain (0-99999). Using dummy variables to represent categorical features is the recommended practice in regression models.

5. If you are using Weka, then convert the CSV into the appropriate ARFF format. If you are using Python, then I recommend looking into Numpy's loadtxt() function.
   I only used Weka to plot features in the train data set (see Appendix B).

6. If you are using Python/SKlearn, then implement the learning algorithm.
   Learning algorithm was implemented with Python and sklearn.
   Code is include in Appendix C and also uploaded with PA#1 assignment report in Canvas (file predict_adult.py)

Please refer to code in Appendix C, for example, function "learn_logit"

8. Evaluate the learned model on the test set.
   a) If you used a validation set to tune your hyper-parameters, then make sure to use the model with the best performance on the test set.
      Each model is tuned with 10 kfold validation and tested on the train dataset validation partition (20% of training data) as explained under the answer for Question 2. Tuning parameters are also described under Question 2 above. Program output is included in Appendix A.
      Note that instead of running only the best model on the test dataset (file "adult.test") we ran all models against it, always using the best hyper parameter setting identified during training and tuning. This was done for illustrative purposes.
      As explained in the answer to Question 2, prior to running the learned model against the test dataset we learn the model against the entire training set with the proper hyper parameter setting.
      Training results are listed in the program output in Appendix A under the title "TRAIN VALIDATION PARTITION".
      Training results are listed in the program output in Appendix A under the title "TEST DATASET".

9. Report the performance of the model.
   a) Make sure to use an appropriate performance measure (ie. accuracy vs. ROC vs. F1-score). Explain why you chose that performance measure.
      Since the proportion of adults with salary >$50k vs. salary <=$50 is slightly unbalanced (approximately 29% proportion with salary >$50k) we chose to use F1-score rather than accuracy to evaluate performance. We tuned all algorithms to capture the hyper parameter yielding the best F1-score.

   b) Explain the effect different hyper-parameters have on the performance of the algorithm. For example, what effect does the depth of the tree have on the performance of the decision tree?
      For Naïve Bayes

Algorithm	Parameter	Effect
Naïve Bayes	None known in sklearn	N/A. Could not find tuning parameter for Naïve Bayes in sklearn.
Decision Tree	Tree Depth	Higher depth will increase overfitting; lower depth will increase bias.
KNN	K neighbors	Lower K will increase overfitting; higher K will increase bias.
Logistic Regression	Penalty (l1, l2) and C	Both parameters impact regularization of the regression function and reduce overfitting.

10. Briefly compare the performance of the different learning algorithms on the test set and make some hypothesizes about why you might have observed these differences.

The table below summarizes our results. We attribute performance differences among the classifiers to undetermined characteristics of the dataset features and their relationship to each other and the target label.
Best performance in each metric in the training set is highlighted in light blue.
Best performance in each metric in the test set is highlighted in light green.
Bold font indicates an improvement or no performance change for the metric on the test dataset (vs. train dataset).

Metric	Naïve Bayes		Decision Tree (depth=9)		KNN (K=13)		Logistic Regression (l1, C=1)	
	Train	Test	Train	Test	Train	Test	Train	Test
Accuracy	0.788	**0.796**	0.854	0.842	0.811	0.788	0.848	**0.848**
Error	0.212	0.204	0.146	0.158	0.189	0.212	0.152	**0.152**
Precision	0.603	0.591	0.675	0.623	0.612	0.500	0.667	0.661
Recall	0.648	0.600	0.607	0.533	0.599	0.432	0.614	0.604
F1-Score	0.564	**0.582**	0.760	0.750	0.625	0.595	0.730	**0.731**
ROC AUC	0.741	0.730	0.772	0.738	0.740	0.668	0.770	0.766

Logistic regression clearly offered the best performance on the test dataset. Decision tree offered the best performance on the training dataset. Most algorithms show overfitting with the exception of Naïve Bayes (accuracy and F1-Score improved) and, to a certain extent Logistic Regression (accuracy unchanged and F1-Score improved). The decision tree performance decreased noticeably on the test dataset demonstrating the overfitting tendency of this algorithm. Interestingly, we rerun our program forcing the maximum depth of the decision tree to 8 and got better results in the train test partition and the test dataset. The results are replicated below. They seem to suggest that it is worth decreasing the max tree depth hyper parameter further after tuning. We attempted a similar strategy for KNN, where we increased K in an attempt to reduce overfitting after tuning, but the results were disappointing.

Performance scores highlighted in dark green indicate scores superior to Logistic Regression scores. In a production setting we would select between Logistic Regression and Decision Tree (max depth=8) based on cost analysis of specific misclassification scenarios (for example, FP vs. FN cost).

Metric	Decision Tree (depth=9)		Decision Tree (depth=8)	
	Train	Test	Train	Test
Accuracy	0.854	0.842	0.848	0.851
Error	0.146	0.158	0.152	0.149
Precision	0.675	0.623	0.635	0.641
Recall	0.607	0.533	0.533	0.542
F1-Score	0.760	0.750	0.786	0.784
ROC AUC	0.772	0.738	0.742	0.747

On a final note, we point out that our results are comparable to the results reported on the web site where the datasets were published (https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.names). The best result reported on the site shows accuracy=0.8595 for "FSS Naïve Bayes". That is a mere 1% improvement over our best accuracy result with the Decision Tree of max depth 8.

References:

[1] McKinney, Wes. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. " O'Reilly Media, Inc.", 2012.

[2] James, Gareth, et al. *An introduction to statistical learning*. Vol. 6. New York: springer, 2013.

[3] Raschka, Sebastian. *Python machine learning*. Packt Publishing Ltd, 2015.

[4] Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*. Vol. 1. Springer, Berlin: Springer series in statistics, 2001.

[5] Downey, Allen. *Think Python*. " O'Reilly Media, Inc.", 2012.

INFO-I526 APPLIED MACHINE LEARNING – SPRING 2017
Carlos Sathler (cssathler@gmail.com)

APPENDIX A

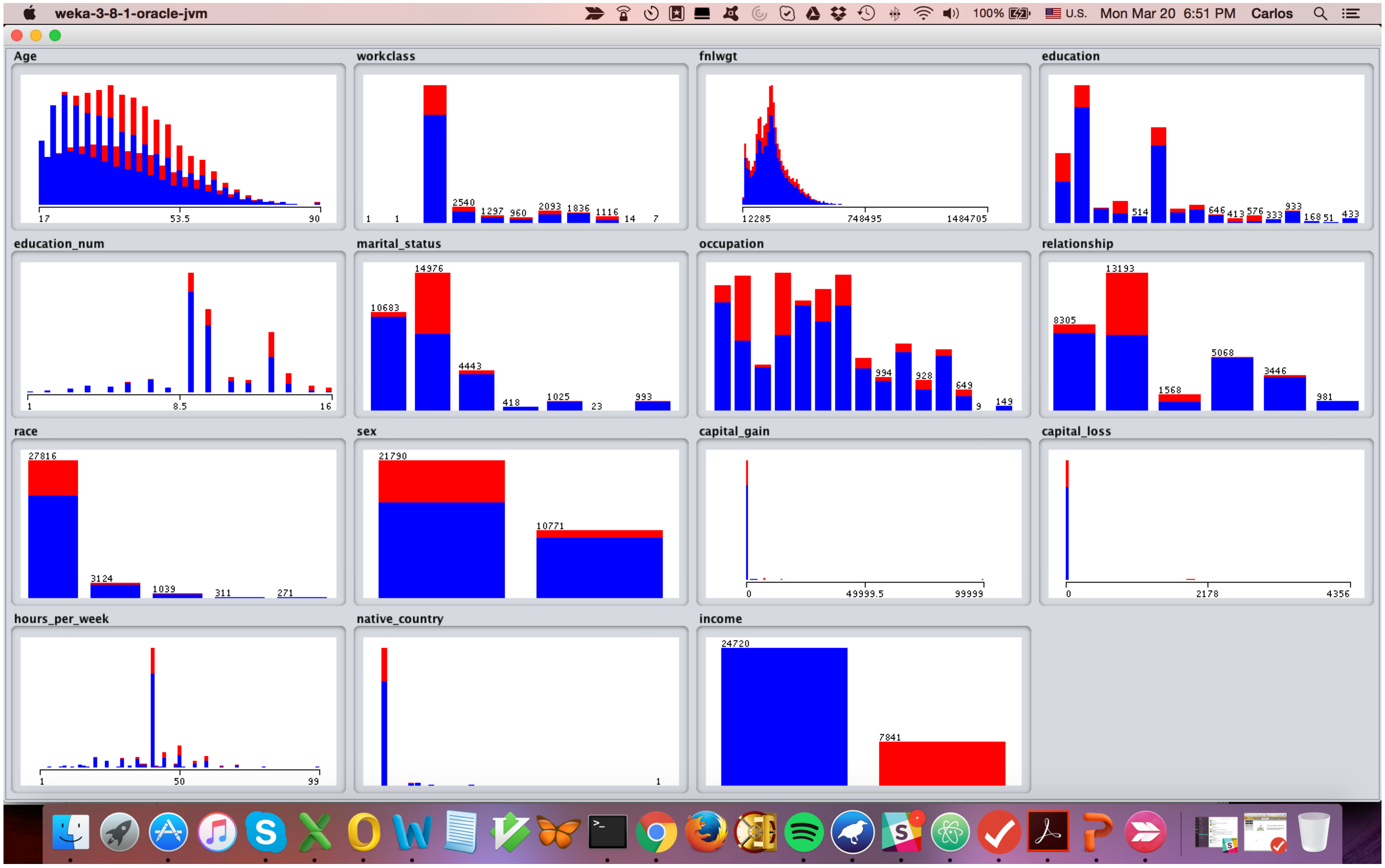
```
RESULTS FOR NAIVE BAYES CLASSIFIER
----------------------------------

           TRAIN VALIDATION PARTITION
           Accuracy........: 0.788
           Precision Score.: 0.603
           Recall Score....: 0.648
           F1 Score........: 0.564
           ROC AUC.........: 0.741

           Confusion Matrix:
           +-----------------------------+
           |         Prediction          |
           +-----------------------------+
           |      0       |       1      |
       +---|-----------------------------|
       | 0 |     3780     |      752     |
 Truth +   |-----------------------------|
       | 1 |      528     |      973     |
       +---+-----------------------------+

           TEST DATASET
           Accuracy........: 0.796
           Precision Score.: 0.591
           Recall Score....: 0.600
           F1 Score........: 0.582
           ROC AUC.........: 0.730

           Confusion Matrix:
           +-----------------------------+
           |         Prediction          |
           +-----------------------------+
           |      0       |       1      |
       +---|-----------------------------|
       | 0 |     9767     |      1593    |
 Truth +   |-----------------------------|
       | 1 |     1479     |      2221    |
       +---+-----------------------------+

RESULTS FOR DECISION TREE CLASSIFIER
------------------------------------
Best tree depth = 9

           TRAIN VALIDATION PARTITION
           Accuracy........: 0.855
           Precision Score.: 0.676
           Recall Score....: 0.608
           F1 Score........: 0.760
           ROC AUC.........: 0.772

           Confusion Matrix:
           +-----------------------------+
           |         Prediction          |
           +-----------------------------+
           |      0       |       1      |
       +---|-----------------------------|
       | 0 |     4244     |      288     |
```

```
       +---+----------------------------+

           TEST DATASET
           Accuracy........: 0.841
           Precision Score.: 0.624
           Recall Score....: 0.535
           F1 Score........: 0.748
           ROC AUC.........: 0.738

           Confusion Matrix:
           +----------------------------+
           |         Prediction         |
           +----------------------------+
           |      0       |      1      |
       +---|----------------------------|
       | 0 |    10692     |     668     |
 Truth +   |----------------------------|
       | 1 |     1720     |     1980    |
       +---+----------------------------+

RESULTS FOR KNN CLASSIFIER
--------------------------
Best neighbors value = 13

           TRAIN VALIDATION PARTITION
           Accuracy........: 0.811
           Precision Score.: 0.612
           Recall Score....: 0.599
           F1 Score........: 0.625
           ROC AUC.........: 0.740

           Confusion Matrix:
           +----------------------------+
           |         Prediction         |
           +----------------------------+
           |      0       |      1      |
       +---|----------------------------|
       | 0 |     3993     |      539    |
 Truth +   |----------------------------|
       | 1 |      602     |      899    |
       +---+----------------------------+

           TEST DATASET
           Accuracy........: 0.788
           Precision Score.: 0.500
           Recall Score....: 0.432
           F1 Score........: 0.595
           ROC AUC.........: 0.668

           Confusion Matrix:
           +----------------------------+
           |         Prediction         |
           +----------------------------+
           |      0       |      1      |
       +---|----------------------------|
       | 0 |    10273     |     1087    |
 Truth +   |----------------------------|
       | 1 |     2103     |     1597    |
       +---+----------------------------+
```

```
RESULTS FOR LOGISTIC REGRESSION CLASSIFIER
------------------------------------------
Best penalty= l1
Best C value= 1.0

           TRAIN VALIDATION PARTITION
           Accuracy........: 0.848
           Precision Score.: 0.667
           Recall Score....: 0.614
           F1 Score........: 0.730
           ROC AUC.........: 0.770

           Confusion Matrix:
           +----------------------------+
           |         Prediction         |
           +----------------------------+
           |      0      |      1       |
       +---|----------------------------|
       | 0 |    4191     |     341      |
 Truth +   |----------------------------|
       | 1 |     579     |     922      |
       +---+----------------------------+

           TEST DATASET
           Accuracy........: 0.848
           Precision Score.: 0.661
           Recall Score....: 0.604
           F1 Score........: 0.731
           ROC AUC.........: 0.766

           Confusion Matrix:
           +----------------------------+
           |         Prediction         |
           +----------------------------+
           |      0      |      1       |
       +---|----------------------------|
       | 0 |   10536     |     824      |
 Truth +   |----------------------------|
       | 1 |    1465     |    2235      |
       +---+----------------------------+
```

INFO-I526 APPLIED MACHINE LEARNING – SPRING 2017
Carlos Sathler (cssathler@gmail.com)

APPENDIX B



INFO-I526 APPLIED MACHINE LEARNING – SPRING 2017

Carlos Sathler (cssathler@gmail.com)

APPENDIX C

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.neighbors import KNeighborsClassifier
from sklearn import tree
from sklearn.metrics import precision_score, recall_score, f1_score, confusion_matrix
from sklearn.metrics import roc_curve, roc_auc_score
from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import scale


#----------------------------------------------------------------------------------------------------
# show_detailed_results
# Show detailed score results for algo, along with confustion matrix
# in: accuracy score, true target values, predicitons
# out:
#
def show_detailed_results(msg, ac, y_test, y_pred):

    f1  = f1_score(y_test, y_pred)
    rc  = recall_score(y_test, y_pred)
    pr  = precision_score(y_test, y_pred)
    cm  = confusion_matrix(y_test, y_pred)
    roc = roc_auc_score(y_test, y_pred)

    print('\n           ' + msg)
    print("           Accuracy........: %.3f" % ac)
    print("           Precision Score.: %.3f" % f1)
    print("           Recall Score....: %.3f" % rc)
    print("           F1 Score........: %.3f" % pr)
    print("           ROC AUC.........: %.3f" % roc)
    print("\n           Confusion Matrix:")

    print("           +----------------------------+")
    print("           |         Prediction         |")
    print("           +----------------------------+")
    print("           |      0      |       1      |")
    print("       +---|----------------------------|")
    print("       | 0 |    %5d    |     %5d    |" % (cm[0][0],cm[0][1]))
    print(" Truth +   |----------------------------|")
    print("       | 1 |    %5d    |     %5d    |" % (cm[1][0],cm[1][1]))
    print("       +---+----------------------------+")


#----------------------------------------------------------------------------------------------------
# gets dictionary and swaps keys with values
# in : my_dict_in
# out: my_dict_out
#
def invert_dict(my_dict_in):
    my_dict_out = {}
    for x, y in my_dict_in.iteritems(): my_dict_out[y] = float(x)
    return my_dict_out

#----------------------------------------------------------------------------------------------------
# loads dataset
# in:  none
# out: dataset as dataframe
#
def load_data(fname):
    # define column names for data import
    df_col_names = ['age', 'workclass', 'fnlwgt', 'education', 'education_num', 'marital_status',
                    'occupation', 'relationship', 'race', 'sex', 'capital_gain', 'capital_loss',
                    'hours_per_week', 'native_country', 'income']
    df_col_dtype = {'age': np.float64, 'fnlwgt': np.float64, 'education_num': np.float64,
                    'capital_gain': np.float64, 'capital_loss': np.float64, 'hours_per_week':
np.float64}
    # read data
    df = pd.read_csv(fname, index_col=None, header=None, na_values=' ?',
                     names=df_col_names, dtype=df_col_dtype)
```

```

#----------------------------------------------------------------------------------------------------
# general precessing
# Performs the following preprocessing to benefit all algorithms
# 1 - converts all columns in dataset to float
#     all categorical and ordinal features are converted
# 2 - remove rows with nan's
# 3 - remove redundant features
#
# in:  dataset as dataframe
# out: tuple with three elements
#      1 - source dataframe w/o nan's and redundant features
#      2 - source dataframe used to create np.arrays - all features as floats
#      3 - dictionary with mapping between original feature values and new number values
#
def general_preprocess(df):

    # found 2399 rows with NaN in the following columns: workclass, occupation, native_country
    # cannot think of a way to impute values, so will dropna
    df.dropna(inplace=True)

    # drop education since we have education_num
    df.drop('education', axis=1, inplace=True)

    df_source = df.copy()

    col_map = {}

    # create dictionary objects for each column containing nominal or ordinal values
    workclass_dict      = pd.Series(df.workclass.unique()).to_dict()
    marital_status_dict = pd.Series(df.marital_status.unique()).to_dict()
    occupation_dict     = pd.Series(df.occupation.unique()).to_dict()
    relationship_dict   = pd.Series(df.relationship.unique()).to_dict()
    race_dict           = pd.Series(df.race.unique()).to_dict()
    sex_dict            = pd.Series(df.sex.unique()).to_dict()
    native_country_dict = pd.Series(df.native_country.unique()).to_dict()
    income_dict         = pd.Series(df.income.unique()).to_dict()

    # create inverted dictionary objects to update dataframe columns
    workclass_dict_inv      = invert_dict( workclass_dict      )
    marital_status_dict_inv = invert_dict( marital_status_dict )
    occupation_dict_inv     = invert_dict( occupation_dict     )
    relationship_dict_inv   = invert_dict( relationship_dict   )
    race_dict_inv           = invert_dict( race_dict           )
    sex_dict_inv            = invert_dict( sex_dict            )
    native_country_dict_inv = invert_dict( native_country_dict )
    income_dict_inv         = invert_dict( income_dict         )

    col_map['workclass']      = [ workclass_dict, workclass_dict_inv ]
    col_map['marital_status'] = [ marital_status_dict, marital_status_dict_inv ]
    col_map['occupation']     = [ occupation_dict, occupation_dict_inv ]
    col_map['relationship']   = [ relationship_dict, relationship_dict_inv ]
    col_map['race']           = [ race_dict, race_dict_inv ]
    col_map['sex']            = [ sex_dict, sex_dict_inv ]
    col_map['native_country'] = [ native_country_dict, native_country_dict_inv ]
    col_map['income']         = [ income_dict, income_dict_inv ]

    # convert each nominal column value to a number using inverted dictionaries
    df.workclass.replace(      workclass_dict_inv      , inplace=True )
    df.marital_status.replace( marital_status_dict_inv , inplace=True )
    df.occupation.replace(     occupation_dict_inv     , inplace=True )
    df.relationship.replace(   relationship_dict_inv   , inplace=True )
    df.race.replace(           race_dict_inv           , inplace=True )
    df.sex.replace(            sex_dict_inv            , inplace=True )
    df.native_country.replace( native_country_dict_inv , inplace=True )
    df.income.replace(         income_dict_inv         , inplace=True )

    return (df_source, df, col_map)

#----------------------------------------------------------------------------------------------------
# Get percentile using global percentile list
# Used by discretize function
# in: value from dataframe continuous feature
# out: return percentile corresponding to value from dataframe column using global percentiles list
#
```

```
    elif (x <= percentiles[1]): return 2.0
    elif (x <= percentiles[2]): return 3.0
    elif (x <= percentiles[3]): return 4.0
    elif (x <= percentiles[4]): return 5.0
    elif (x <= percentiles[5]): return 6.0
    elif (x <= percentiles[6]): return 7.0
    elif (x <= percentiles[7]): return 8.0
    elif (x <= percentiles[8]): return 9.0
    else: return 10.0

#-----------------------------------------------------------------------------------------
# Discretize
# Replace continous feature with discrete value from 1-10 corresponding to it's percentile
# in: dataframe and name of the column to be discretize
#
def discretize(df, col_name):

    # create list with percentiles
    global percentiles
    percentiles = []
    x = []
    for i in range(1,11,1): x.append(float(i)/10)
    percentiles = df[col_name].quantile(x).tolist()

    # discretize column
    df[col_name] = df[col_name].apply(lambda x: getpercentile(x))

    return df

#-----------------------------------------------------------------------------------------
# logit_preprocess
# Pre-processing for logistic regression algo
# will do the following:
# - standardize continuous variables
# - discretize income variable
# - create dummy variable for each categorial feature
# in: dataframe not ready for logit algo
# out: dataframe ready for logit algo
#
def logit_preprocess(df):

    # standardize continous variables
    df.age            = scale(df.age)
    df.fnlwgt         = scale(df.fnlwgt)
    df.education_num  = scale(df.education_num)
    df.capital_gain   = scale(df.capital_gain)
    df.capital_loss   = scale(df.capital_loss)
    df.hours_per_week = scale(df.hours_per_week)

    # discretize income variable
    col_map = {}
    income_dict       = pd.Series(df.income.unique()).to_dict()
    income_dict_inv   = invert_dict( income_dict )
    col_map['income'] = [ income_dict, income_dict_inv ]
    df.income.replace(income_dict_inv, inplace=True )

    # create dummy variables for categorical features
    df_dummies = pd.get_dummies(df)

    return df_dummies


#-----------------------------------------------------------------------------------------
# nbayes_preprocess
# Naive Bayes specific preprocessing
# in: dataframe not ready for Naive Bayes algo
# out: dataframe ready for Naive Bayes algo
#
def nbayes_preprocess(df):

    # Discretize all continous columns
    df = discretize( df, 'age'            )
    df = discretize( df, 'fnlwgt'         )
    df = discretize( df, 'capital_gain'   )
    df = discretize( df, 'capital_loss'   )
```

```python


#----------------------------------------------------------------------------------------------------
# Learn naive_bayes
# in:  dataframe ready for naive bayes algo
# out:
#
def learn_naive_bayes(df, df_TEST):

    # partition dataset
    # will tune using stratified kfold and get final score on test set
    X_train, X_test, y_train, y_test = train_test_split(
           df.iloc[:,:13].values, df.iloc[:,13:14].values, test_size=0.2, random_state=0)

    # create nvaive bayes classifier object
    clf = GaussianNB()

    # will train model using kfold validation and compute average score
    scores = []
    kf = KFold(n_splits=10)

    # split "Generate indices to split data into training and test set" test = validation
    for train_idx, valid_idx in kf.split(X_train, y_train):

        # load train and validation partition for this iteration of score computing
        X_train_part = X_train[np.ravel(train_idx)]
        y_train_part = y_train[np.ravel(train_idx)]
        X_valid_part = X_train[np.ravel(valid_idx)]
        y_valid_part = y_train[np.ravel(valid_idx)]

        # learn/fit model for this fold
        clf = clf.fit( X_train_part, y_train_part)

        # calculate f1 score
        y_pred = clf.predict( X_valid_part)
        scores.append( f1_score(y_valid_part, y_pred) )

    # display results
    #
    print('\nRESULTS FOR NAIVE BAYES CLASSIFIER')
    print('----------------------------------')

    # first results on training dataset - validation partition
    #
    ac = clf.score( X_test, y_test )
    y_pred = clf.predict(X_test)
    show_detailed_results('TRAIN VALIDATION PARTITION', ac, y_test, y_pred)

    # then results on test dataset
    #
    X_train = df.copy()
    X_train.drop('income', axis=1, inplace=True)
    y_train = df.income
    clf = clf.fit( X_train.values, y_train.values )
    #
    X_TEST = df_TEST.copy()
    X_TEST.drop('income', axis=1, inplace=True)
    y_TEST = df_TEST.income
    ac = clf.score( X_TEST, y_TEST)
    y_pred_TEST = clf.predict(X_TEST)
    show_detailed_results('TEST DATASET', ac, y_TEST, y_pred_TEST)



#----------------------------------------------------------------------------------------------------
# Learn decision tree
# in:  dataframe ready for decision tree algo
# out:
#
def learn_decision_tree(df, df_TEST):

    # partition dataset
    # will tune using stratified kfold and get final score on test set
    X_train, X_test, y_train, y_test = train_test_split(
           df.iloc[:,:13].values, df.iloc[:,13:14].values, test_size=0.2, random_state=0)
```

```python

    for parm_value in np.arange(15)+1:

        # create nvaive bayes classifier object with max_depth parameter
        clf = tree.DecisionTreeClassifier(max_depth=parm_value)

        # will train model using kfold validation and compute average score
        scores = []
        kf = KFold(n_splits=10)

        # split "Generate indices to split data into training and test set" test = validation
        for train_idx, valid_idx in kf.split(X_train, y_train):

            # load train and validation partition for this iteration of score computing
            X_train_part = X_train[np.ravel(train_idx)]
            y_train_part = y_train[np.ravel(train_idx)]
            X_valid_part = X_train[np.ravel(valid_idx)]
            y_valid_part = y_train[np.ravel(valid_idx)]

            # learn/fit model for this fold
            clf = clf.fit(X_train_part, np.ravel(y_train_part))

            # calculate f1 score to select best hyperparameter
            y_pred = clf.predict( X_valid_part)
            scores.append( f1_score(y_valid_part, y_pred) )

        # track best score and corresponding hyperparameter value
        if best_score < np.mean(scores):
            best_score = np.mean(scores)
            best_parm  = parm_value

    # repeat learning on full train partition with best parm value
    clf = tree.DecisionTreeClassifier(max_depth=best_parm)
    clf = clf.fit( X_train, np.ravel(y_train))

    # calculate test scores
    #
    print('\nRESULTS FOR DECISION TREE CLASSIFIER')
    print('-----------------------------------')
    print("Best tree depth = %i" % best_parm)

    # first results on training dataset - validation partition
    #
    ac = clf.score( X_test, y_test )
    y_pred = clf.predict(X_test)
    show_detailed_results('TRAIN VALIDATION PARTITION', ac, y_test, y_pred)

    # then results on test dataset
    #
    X_train = df.copy()
    X_train.drop('income', axis=1, inplace=True)
    y_train = df.income
    clf = clf.fit( X_train.values, y_train.values )
    #
    X_TEST = df_TEST.copy()
    X_TEST.drop('income', axis=1, inplace=True)
    y_TEST = df_TEST.income
    ac = clf.score( X_TEST, y_TEST)
    y_pred_TEST = clf.predict(X_TEST)
    show_detailed_results('TEST DATASET', ac, y_TEST, y_pred_TEST)


#----------------------------------------------------------------------------------------------------
# Learn knn algorithm
# in:  dataframe ready for knn algo
# out:
#
def learn_knn(df, df_TEST):

    # partition dataset
    # will tune using stratified kfold and get final score on test set
    X_train, X_test, y_train, y_test = train_test_split(
           df.iloc[:,:13].values, df.iloc[:,13:14].values, test_size=0.2, random_state=0)

    best_parm  = 0
```

```

        # create nvaive bayes classifier object with max_depth parameter
        clf = KNeighborsClassifier(n_neighbors=parm_value)

        # will train model using kfold validation and compute average score
        scores = []
        kf = KFold(n_splits=10)

        # split "Generate indices to split data into training and test set" test = validation
        for train_idx, valid_idx in kf.split(X_train, y_train):

            # load train and validation partition for this iteration of score computing
            X_train_part = X_train[np.ravel(train_idx)]
            y_train_part = y_train[np.ravel(train_idx)]
            X_valid_part = X_train[np.ravel(valid_idx)]
            y_valid_part = y_train[np.ravel(valid_idx)]

            # learn/fit model for this fold
            clf = clf.fit(X_train_part, np.ravel(y_train_part))

            # calculate f1 score to select best hyperparameter
            y_pred = clf.predict( X_valid_part)
            scores.append( f1_score(y_valid_part, y_pred) )

        # track best score and corresponding hyperparameter value
        if best_score < np.mean(scores):
            best_score = np.mean(scores)
            best_parm  = parm_value

    # repeat learning on full train partition with best parm value
    clf = KNeighborsClassifier(n_neighbors=best_parm)
    clf = clf.fit( X_train, np.ravel(y_train) )

    # calculate test scores
    #
    print('\nRESULTS FOR KNN CLASSIFIER')
    print('-------------------------')
    print("Best neighbors value = %i" % best_parm)

    # first results on training dataset - validation partition
    #
    ac = clf.score( X_test, y_test )
    y_pred = clf.predict(X_test)
    show_detailed_results('TRAIN VALIDATION PARTITION', ac, y_test, y_pred)

    # then results on test dataset
    #
    X_train = df.copy()
    X_train.drop('income', axis=1, inplace=True)
    y_train = df.income
    clf = clf.fit( X_train.values, y_train.values )
    #
    X_TEST = df_TEST.copy()
    X_TEST.drop('income', axis=1, inplace=True)
    y_TEST = df_TEST.income
    ac = clf.score( X_TEST, y_TEST)
    y_pred_TEST = clf.predict(X_TEST)
    show_detailed_results('TEST DATASET', ac, y_TEST, y_pred_TEST)


#--------------------------------------------------------------------------------------------------
# Learn logistic regression
# in:  dataframe ready for logit algo
# out:
#
def learn_logit(df, df_TEST):

    X = df.copy()
    X.drop('income', axis=1, inplace=True)
    y = df.income

    # partition dataset
    # will tune using stratified kfold and get final score on test set
    X_train, X_test, y_train, y_test = train_test_split( X.values, y.values,
                                                test_size=0.2, random_state=0)
```

```
    l = ['l1'] * 5 + ['l2'] * 5
    C = [1.0, 10.0, 100.0, 1000.0, 1000000.0] * 2

    for l_, C_ in zip(l,C):

        # create nvaive bayes classifier object with max_depth parameter
        clf = LogisticRegression(penalty=l_, C=C_)

        # will train model using kfold validation and compute average score
        scores = []
        kf = KFold(n_splits=10)

        # split "Generate indices to split data into training and test set" test = validation
        for train_idx, valid_idx in kf.split(X_train, y_train):

            # load train and validation partition for this iteration of score computing
            X_train_part = X_train[np.ravel(train_idx)]
            y_train_part = y_train[np.ravel(train_idx)]
            X_valid_part = X_train[np.ravel(valid_idx)]
            y_valid_part = y_train[np.ravel(valid_idx)]

            # learn/fit model for this fold
            clf = clf.fit(X_train_part, np.ravel(y_train_part))

            # calculate f1 score to select best hyperparameter
            y_pred = clf.predict( X_valid_part)
            scores.append( f1_score(y_valid_part, y_pred) )

        # track best score and corresponding hyperparameter value
        if best_score < np.mean(scores):
            best_score = np.mean(scores)
            best_parm  = [ l_, C_ ]

    # repeat learning on full train partition with best parm value
    clf = LogisticRegression(penalty=best_parm[0], C=best_parm[1])
    clf = clf.fit( X_train, np.ravel(y_train) )

    # calculate test scores
    #
    print('\nRESULTS FOR LOGISTIC REGRESSION CLASSIFIER')
    print('-----------------------------------------')
    print("Best penalty= " + best_parm[0])
    print("Best C value= " + str(best_parm[1]))

    # first results on training dataset - validation partition
    #
    ac = clf.score( X_test, y_test )
    y_pred = clf.predict(X_test)
    show_detailed_results('TRAIN VALIDATION PARTITION', ac, y_test, y_pred)

    # then results on test dataset
    #
    X_train = df.copy()
    X_train.drop('income', axis=1, inplace=True)
    # removing column which doesn't exist in TEST dataset
    X_train.drop('native_country_ Holand-Netherlands', axis=1, inplace=True)
    y_train = df.income
    clf = clf.fit( X_train.values, y_train.values )
    #
    X_TEST = df_TEST.copy()
    X_TEST.drop('income', axis=1, inplace=True)
    y_TEST = df_TEST.income
    ac = clf.score( X_TEST, y_TEST)
    y_pred_TEST = clf.predict(X_TEST)
    show_detailed_results('TEST DATASET', ac, y_TEST, y_pred_TEST)



########################################################################################
#
if __name__ == '__main__':

    # load data
    df = load_data('adult.data')
```

```
    # called "general" because all algorithms will have this preprocessing in common
    # - get df_source back without nan rows and without redundant features
    # - get df_floats with all categorical and nominal features converted to float
    # - get dictionary with mapping info for each converted field
    #
    (df, df_floats, df_col_map) = general_preprocess( df.copy() )
    (df_TEST, df_floats_TEST, df_col_map_TEST) = general_preprocess( df_TEST.copy() )

    # performs pre-processing for naive bayes algorithm
    # - get df_nbayes with continous features properly discretized
    df_nbayes = nbayes_preprocess( df_floats.copy() )
    df_nbayes_TEST = nbayes_preprocess( df_floats_TEST.copy() )
    learn_naive_bayes( df_nbayes, df_nbayes_TEST )

    # performs pre-processing for decision tree algorithm
    # same as naive bayes
    df_dtree = df_nbayes.copy()
    df_dtree_TEST = df_nbayes_TEST.copy()
    #learn_decision_tree( df_dtree, df_dtree_TEST )
    learn_decision_tree( df_floats, df_floats_TEST )     # performed better with continous features

    # performs pre-processing for knn algorithm
    # will use discretized functions to increase clustering of datapoints
    learn_knn( df_nbayes, df_nbayes_TEST )               # performed better with discretized features
    # also trying using continous feature values
    df_knn = df_floats.copy()
    df_knn_TEST = df_floats_TEST.copy()
    #learn_knn( df_knn, df_knn_TEST )

    # performs pre-processing for logistic regression algorithm
    df_logit = logit_preprocess( df.copy() )
    df_logit_TEST = logit_preprocess( df_TEST.copy() )
    learn_logit( df_logit, df_logit_TEST )
```