UNIVERSITY OF MUMBAI

DEPARTMENT OF COMPUTER SCIENCE

M.Sc. Computer Science with Spl. in Data Science – Semester III

Advanced Machine Learning

JOURNAL

2022-2023

Seat No. 30283

मुंबई विद्यापीठ
University of Mumbai
Re-accredited with A++ Grade
(CGPA 3.65) by NAAC (3rd Cycle 2021)

UNIVERSITY OF MUMBAI

DEPARTMENT OF COMPUTER SCIENCE

CERTIFICATE

This is to certify that the work entered in this journal was done in the University Department of Computer Science laboratory by Mr./~~Ms.~~ **Sumon Singh** Seat No. **30283** for the course of M.Sc. Computer Science with Spl. in Data Science - Semester III (CBCS) (Revised) during the academic year 2022-2023 in a satisfactory manner.

_____ _____

Subject In-charge **Head of Department**

_____

External Examiner

Index

Practical - 1

AIM : Implement Logistic Regression

Load Required Libraries

In [1]:

```
import pandas as pd
import numpy as np
from sklearn import preprocessing
import matplotlib.pyplot as plt
plt.rc("font", size=14)
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
import seaborn as sns
sns.set(style="white")
sns.set(style="whitegrid", color_codes=True)
```

The dataset comes from the UCI Machine Learning repository, and it is related to direct marketing campaigns (phone calls) of a Portuguese banking institution. The classification goal is to predict whether the client will subscribe (1/0) to a term deposit (variable y).

The dataset provides the bank customers' information. It includes 41,188 records and 21 fields.

Load Dataset

In [2]:

```
data = pd.read_csv('/home/sumon/Documents/Datasets/banking.csv')
```

In [3]:

```
data = data.dropna()
print(data.shape)
print(list(data.columns))
```

```
(41188, 21)
['age', 'job', 'marital', 'education', 'default', 'housing', 'loan', 'contact', 'month', 'day_of_week',
'duration', 'campaign', 'pdays', 'previous', 'poutcome', 'emp_var_rate', 'cons_price_idx', 'cons_conf_i
dx', 'euribor3m', 'nr_employed', 'y']
```

In [4]:

```
data['education'].unique()
```

Out[4]:

```
array(['basic.4y', 'unknown', 'university.degree', 'high.school',
       'basic.9y', 'professional.course', 'basic.6y', 'illiterate'],
      dtype=object)
```

Let us group "basic.4y", "basic.9y" and "basic.6y" together and call them "basic".

In [5]:

```
data['education']=np.where(data['education'] =='basic.9y', 'Basic', data['education'])
data['education']=np.where(data['education'] =='basic.6y', 'Basic', data['education'])
data['education']=np.where(data['education'] =='basic.4y', 'Basic', data['education'])
```

Check the columns after grouping

In [6]:

```
data['education'].unique()
```

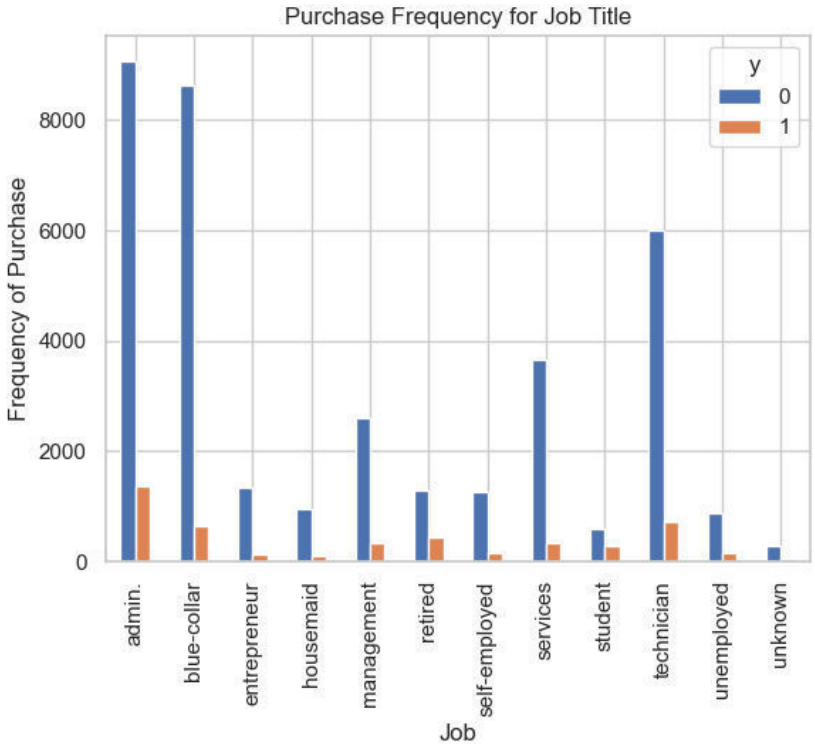
Out[6]:

```
array(['Basic', 'unknown', 'university.degree', 'high.school',
       'professional.course', 'illiterate'], dtype=object)
```

Visualization

In [7]:

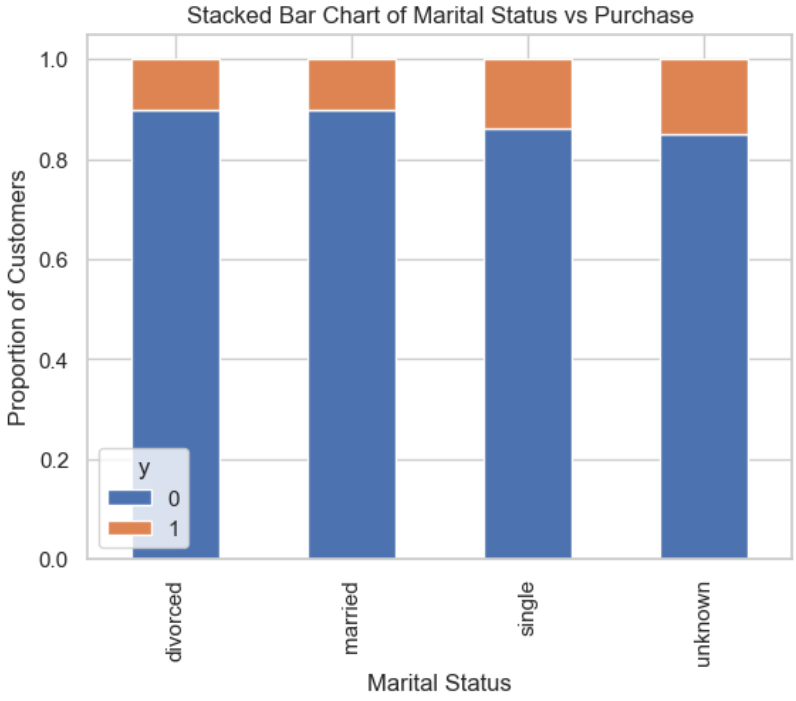
```
%matplotlib inline
pd.crosstab(data.job,data.y).plot(kind='bar')
plt.title('Purchase Frequency for Job Title')
plt.xlabel('Job')
plt.ylabel('Frequency of Purchase')
plt.savefig('purchase_fre_job')
```



The frequency of purchase of the deposit depends a great deal on the job title. Thus, the job title can be a good predictor of the outcome variable.

In [8]:

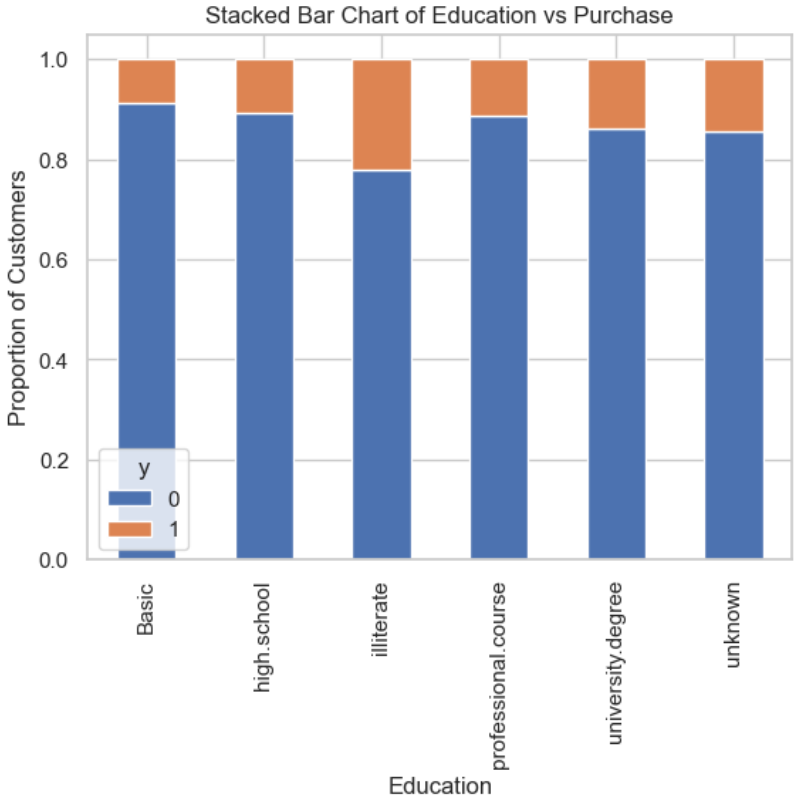
```
table=pd.crosstab(data.marital,data.y)
table.div(table.sum(1).astype(float), axis=0).plot(kind='bar',
                                                   stacked=True)
plt.title('Stacked Bar Chart of Marital Status vs Purchase')
plt.xlabel('Marital Status')
plt.ylabel('Proportion of Customers')
plt.savefig('mariral_vs_pur_stack')
```



The marital status does not seem a strong predictor for the outcome variable.

In [9]:

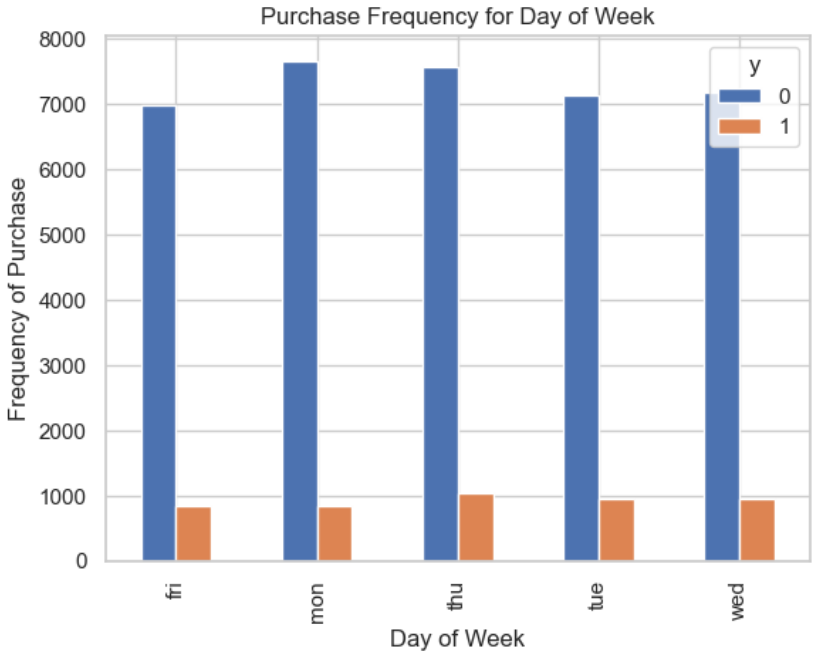
```
table=pd.crosstab(data.education,data.y)
table.div(table.sum(1).astype(float), axis=0).plot(kind='bar',
                                                   stacked=True)
plt.title('Stacked Bar Chart of Education vs Purchase')
plt.xlabel('Education')
plt.ylabel('Proportion of Customers')
plt.savefig('edu_vs_pur_stack')
```



Education seems a good predictor of the outcome variable.

In [10]:

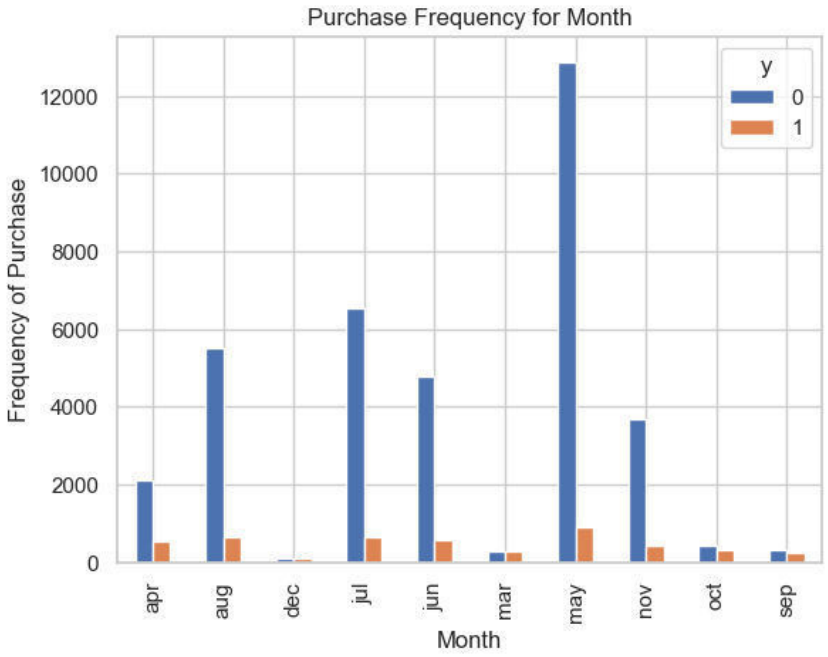
```
pd.crosstab(data.day_of_week,data.y).plot(kind='bar')
plt.title('Purchase Frequency for Day of Week')
plt.xlabel('Day of Week')
plt.ylabel('Frequency of Purchase')
plt.savefig('pur_dayofweek_bar')
```



Day of week may not be a good predictor of the outcome.

In [11]:

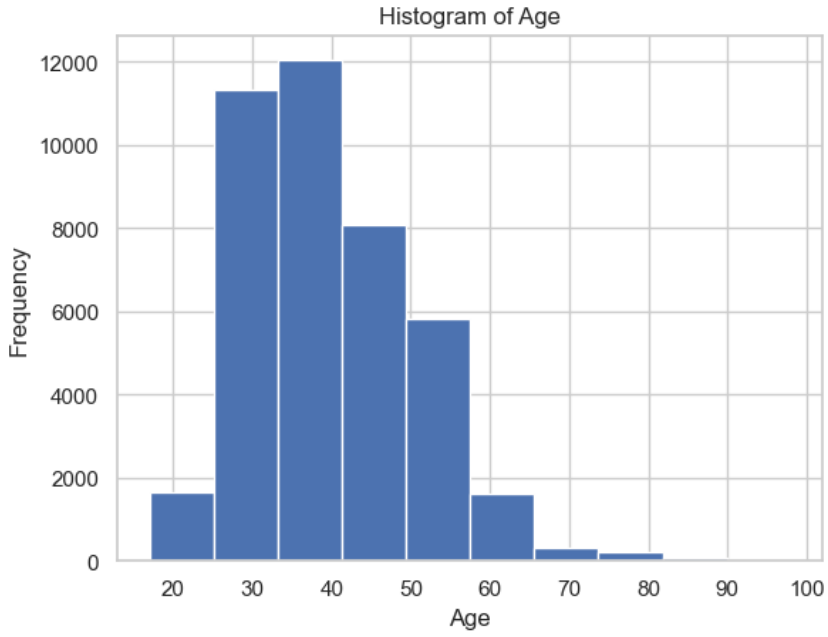
```
pd.crosstab(data.month,data.y).plot(kind='bar')
plt.title('Purchase Frequency for Month')
plt.xlabel('Month')
plt.ylabel('Frequency of Purchase')
plt.savefig('pur_fre_month_bar')
```



Month might be a good predictor of the outcome variable.

In [12]:

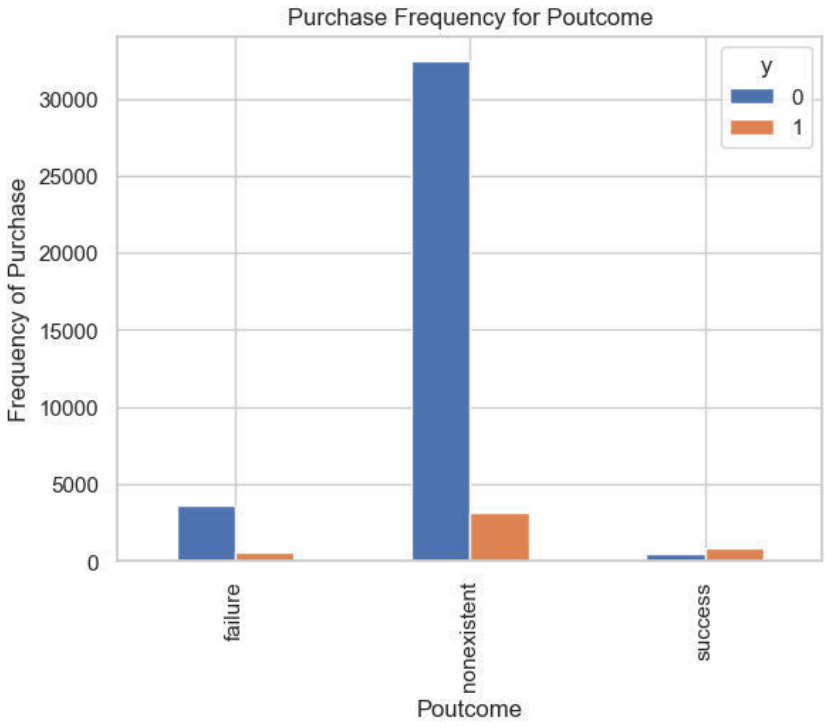
```
data.age.hist()
plt.title('Histogram of Age')
plt.xlabel('Age')
plt.ylabel('Frequency')
plt.savefig('hist_age')
```



Most of the customers of the bank in this dataset are in the age range of 30–40.

In [13]:

```
pd.crosstab(data.poutcome,data.y).plot(kind='bar')
plt.title('Purchase Frequency for Poutcome')
plt.xlabel('Poutcome')
plt.ylabel('Frequency of Purchase')
plt.savefig('pur_fre_pout_bar')
```



Poutcome seems to be a good predictor of the outcome variable.

Create dummy variables

That is variables with only two values, zero and one.

In [14]:

```
cat_vars=['job','marital','education','default','housing',
          'loan','contact','month','day_of_week','poutcome']
for var in cat_vars:
    cat_list='var'+'_'+var
    cat_list = pd.get_dummies(data[var], prefix=var)
    data1=data.join(cat_list)
    data=data1
cat_vars=['job','marital','education','default','housing',
          'loan','contact','month','day_of_week','poutcome']
data_vars=data.columns.values.tolist()
to_keep=[i for i in data_vars if i not in cat_vars]
```

In [15]:

```
data_final=data[to_keep]
data_final.columns.values
```

Out[15]:

```
array(['age', 'duration', 'campaign', 'pdays', 'previous', 'emp_var_rate',
       'cons_price_idx', 'cons_conf_idx', 'euribor3m', 'nr_employed', 'y',
       'job_admin.', 'job_blue-collar', 'job_entrepreneur',
       'job_housemaid', 'job_management', 'job_retired',
       'job_self-employed', 'job_services', 'job_student',
       'job_technician', 'job_unemployed', 'job_unknown',
       'marital_divorced', 'marital_married', 'marital_single',
       'marital_unknown', 'education_Basic', 'education_high.school',
       'education_illiterate', 'education_professional.course',
       'education_university.degree', 'education_unknown', 'default_no',
       'default_unknown', 'default_yes', 'housing_no', 'housing_unknown',
       'housing_yes', 'loan_no', 'loan_unknown', 'loan_yes',
       'contact_cellular', 'contact_telephone', 'month_apr', 'month_aug',
       'month_dec', 'month_jul', 'month_jun', 'month_mar', 'month_may',
       'month_nov', 'month_oct', 'month_sep', 'day_of_week_fri',
       'day_of_week_mon', 'day_of_week_thu', 'day_of_week_tue',
       'day_of_week_wed', 'poutcome_failure', 'poutcome_nonexistent',
       'poutcome_success'], dtype=object)
```

In [16]:

```
from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegression
```

In [17]:

```
data_final_vars=data_final.columns.values.tolist()
y=['y']
X=[i for i in data_final_vars if i not in y]

logreg = LogisticRegression()
```

In [18]:

```
X=data_final[X]
y=data_final[y]
```

In [19]:

```
import statsmodels.api as sm
logit_model=sm.Logit(y,X)
result=logit_model.fit()
print(result.summary2())
```

```
Warning: Maximum number of iterations has been exceeded.
         Current function value: 0.207332
         Iterations: 35
                                    Results: Logit
==========================================================================================
Model:                      Logit                    Pseudo R-squared:         0.411
Dependent Variable:         y                        AIC:                      17181.1442
Date:                       2023-01-07 17:38         BIC:                      17621.0653
No. Observations:           41188                    Log-Likelihood:           -8539.6
Df Model:                   50                       LL-Null:                  -14499.
Df Residuals:               41137                    LLR p-value:              0.0000
Converged:                  0.0000                   Scale:                    1.0000
No. Iterations:             35.0000
------------------------------------------------------------------------------------------
                            Coef.     Std.Err.      z     P>|z|      [0.025       0.975]
------------------------------------------------------------------------------------------
age                          0.0001       0.0024   0.0410 0.9673       -0.0046       0.0048
duration                     0.0047       0.0001  63.1105 0.0000        0.0046       0.0049
campaign                    -0.0402       0.0116  -3.4745 0.0005       -0.0628      -0.0175
```

We can remove the variables with ehe p-values for most of the variables are more than 0.05 to improve the accuracy.

In [20]:

```
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.3,
                                                    random_state=0)
logreg = LogisticRegression()
logreg.fit(X_train, y_train)
```

```
/home/sumon/.local/lib/python3.10/site-packages/sklearn/utils/validation.py:993: DataConversionWarning:
A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples,
), for example using ravel().
  y = column_or_1d(y, warn=True)
/home/sumon/.local/lib/python3.10/site-packages/sklearn/linear_model/_logistic.py:814: ConvergenceWarni
ng: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://scikit-learn.org/stable/module
s/preprocessing.html)
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression (https://scikit-lear
n.org/stable/modules/linear_model.html#logistic-regression)
  n_iter_i = _check_optimize_result(
```

Out[20]:

```
LogisticRegression()
```

In [21]:

```
y_pred = logreg.predict(X_test)
print('Accuracy of logistic regression classifier on test set: {:.2f}'
     .format(logreg.score(X_test, y_test)))
```

```
Accuracy of logistic regression classifier on test set: 0.91
```

In [22]:

```
from sklearn.metrics import confusion_matrix
confusion_matrix = confusion_matrix(y_test, y_pred)
print(confusion_matrix)
```

```
[[10704   277]
 [  802   574]]
```

In [23]:

```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.93      0.97      0.95     10981
           1       0.67      0.42      0.52      1376

    accuracy                           0.91     12357
   macro avg       0.80      0.70      0.73     12357
weighted avg       0.90      0.91      0.90     12357
```

Practical - 2

AIM : Implement RidgeRegression

Load and summarize the housing dataset. Evaluate an ridge regression model on the dataset

Ridge Regression

Linear regression refers to a model that assumes a linear relationship between input variables and the target variable.

With a single input variable, this relationship is a line, and with higher dimensions, this relationship can be thought of as a hyperplane that connects the input variables to the target variable. The coefficients of the model are found via an optimization process that seeks to minimize the sum squared error between the predictions (yhat) and the expected target values (y).

loss = sum i=0 to n (y_i – yhat_i)^2

A problem with linear regression is that estimated coefficients of the model can become large, making the model sensitive to inputs and possibly unstable. This is particularly true for problems with few observations (samples) or less samples (n) than input predictors (p) or variables (so-called p >> n problems).

One approach to address the stability of regression models is to change the loss function to include additional costs for a model that has large coefficients. Linear regression models that use these modified loss functions during training are referred to collectively as penalized linear regression.

One popular penalty is to penalize a model based on the sum of the squared coefficient values (beta). This is called an L2 penalty.

l2_penalty = sum j=0 to p beta_j^2

An L2 penalty minimizes the size of all coefficients, although it prevents any coefficients from being removed from the model by allowing their value to become zero.

Load necessary libraries

In [1]:

```
from pandas import read_csv
from matplotlib import pyplot
from numpy import mean
from numpy import std
from numpy import absolute
from pandas import read_csv
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedKFold
from sklearn.linear_model import Ridge
```

Load dataset

In [2]:

```
url = '/home/sumon/Documents/Datasets/housing.csv'
dataframe = read_csv(url, header=None)
```

Summarize shape

In [3]:

```
print(dataframe.shape)
```

```
(506, 14)
```

Summarize first few lines

In [4]:

```
print(dataframe.head())
```

```
        0     1     2   3      4      5     6       7   8      9     10  \
0  0.00632  18.0  2.31   0  0.538  6.575  65.2  4.0900   1  296.0  15.3
1  0.02731   0.0  7.07   0  0.469  6.421  78.9  4.9671   2  242.0  17.8
2  0.02729   0.0  7.07   0  0.469  7.185  61.1  4.9671   2  242.0  17.8
3  0.03237   0.0  2.18   0  0.458  6.998  45.8  6.0622   3  222.0  18.7
4  0.06905   0.0  2.18   0  0.458  7.147  54.2  6.0622   3  222.0  18.7

       11    12    13
0  396.90  4.98  24.0
1  396.90  9.14  21.6
2  392.83  4.03  34.7
3  394.63  2.94  33.4
4  396.90  5.33  36.2
```

Make a prediction with a ridge regression model on the dataset

In [5]:

```
data = dataframe.values
X, y = data[:, :-1], data[:, -1]
```

The scikit-learn Python machine learning library provides an implementation of the Ridge Regression algorithm via the Ridge class.

The default value for alpha is 1.0 or a full penalty.

Define model

In [6]:

```
model = Ridge(alpha=1.0)
```

Fit model

In [7]:

```
model.fit(X, y)
```

Out[7]:

```
Ridge()
```

Define new data

In [8]:

```
row = [0.00632,18.00,2.310,0,0.5380,6.5750,65.20,4.0900,
       1,296.0,15.30,396.90,4.98]
```

Make a prediction

In [9]:

```
yhat = model.predict([row])
```

Summarize prediction

In [10]:

```
print('Predicted: %.3f' % yhat)
```

```
Predicted: 30.253
```

We can evaluate the Ridge Regression model on the housing dataset using repeated 10-fold cross-validation and report the average mean absolute error (MAE) on the dataset.

Define model evaluation method

In [11]:

```
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
```

Evaluate model

In [12]:

```
scores = cross_val_score(model, X, y, scoring='neg_mean_absolute_error',
                         cv=cv, n_jobs=-1)
```

Force scores to be positive

In [13]:

```
scores = absolute(scores)
print('Mean MAE: %.3f (%.3f)' % (mean(scores), std(scores)))
```

```
Mean MAE: 3.382 (0.519)
```

Modification-2: Tuning Ridge Hyperparameters

the default hyperparameters of alpha=1.0

Instead, it is good practice to test a suite of different configurations and discover what works best for our dataset.

One approach would be to grid search alpha values from perhaps 1e-5 to 100 on a log scale and discover what works best for a dataset. Another approach would be to test values between 0.0 and 1.0 with a grid separation of 0.01. We will try the latter in this case.

In [14]:

```
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RepeatedKFold
from numpy import arange
```

Define model

In [15]:

```
model = Ridge()
```

Define model evaluation method

In [16]:

```
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
```

Define grid

In [17]:

```
grid = dict()
grid['alpha'] = arange(0, 1, 0.01)
```

Define search

In [18]:

```
search = GridSearchCV(model, grid, scoring='neg_mean_absolute_error',
                      cv=cv, n_jobs=-1)
```

Perform the search

In [19]:

```
results = search.fit(X, y)
```

Summarize

In [20]:

```
print('MAE: %.3f' % results.best_score_)
print('Config: %s' % results.best_params_)
```

```
MAE: -3.379
Config: {'alpha': 0.51}
```

Running the example will evaluate each combination of configurations using repeated cross-validation.

Your specific results may vary given the stochastic nature of the learning algorithm. Try running the example a few times.

In this case, we can see that we achieved slightly better results than the default 3.379 vs. 3.382. Ignore the sign; the library makes the MAE negative for optimization purposes.

We can see that the model assigned an alpha weight of 0.51 to the penalty.

Practical - 3

AIM : Implement Adaboost

Load required libraries

In [1]:

```
import pandas as pd
from sklearn.ensemble import AdaBoostClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
```

Read the dataset

In [2]:

```
df = pd.read_csv('/home/sumon/Documents/Datasets/apples_and_oranges.csv')
```

Get the locations

In [3]:

```
X = df.iloc[:, :-1]
y = df.iloc[:, -1]
```

Code to view first 5 records in the data set

In [4]:

```
df.head()
```

Out[4]:

	Weight	Size	Class
0	69	4.39	orange
1	69	4.21	orange
2	65	4.09	orange
3	72	5.85	apple
4	67	4.70	orange

In [5]:

```
X.head()
```

Out[5]:

	Weight	Size
0	69	4.39
1	69	4.21
2	65	4.09
3	72	5.85
4	67	4.70

In [6]:

```
y.head()
```

Out[6]:

```
0    orange
1    orange
2    orange
3     apple
4    orange
Name: Class, dtype: object
```

Split the dataset

In [7]:

```
seed = 1
X_train, X_test, Y_train, Y_test = train_test_split(
    X, y, test_size=0.2, random_state=seed)
```

Initializing Adaboost classifier and fitting the training data

In [8]:

```
adaboost = AdaBoostClassifier(n_estimators=100,
                              base_estimator= None,
                              learning_rate=1, random_state = 1)
adaboost.fit(X_train,Y_train)
```

Out[8]:

```
AdaBoostClassifier(learning_rate=1, n_estimators=100, random_state=1)
```

Predicting the classes for test set

In [9]:

```
Y_pred = adaboost.predict(X_test)
```

In [10]:

```
cm = confusion_matrix(Y_test,Y_pred)
accuracy = float(cm.diagonal().sum())/len(Y_test)
print("\nAccuracy Of AdaBoost For The Given Dataset : ", accuracy)
```

```
Accuracy Of AdaBoost For The Given Dataset :  1.0
```

Practical - 4

AIM : Implement XGBoost

Install xgboost

In [1]:

```
!pip3 install xgboost
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: xgboost in /home/sumon/.local/lib/python3.10/site-packages (1.7.2)
Requirement already satisfied: numpy in /home/sumon/.local/lib/python3.10/site-packages (from xgboost)
(1.23.4)
Requirement already satisfied: scipy in /home/sumon/.local/lib/python3.10/site-packages (from xgboost)
(1.9.3)
```

Load necessary libraries

In [2]:

```
import pandas as pd
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

Load the dataset and decide your X, y values, need to split the data into train/test sets.

train_test_split(*arrays, test_size=None, train_size=None, random_state=None, shuffle=True, stratify=None)

Parameters:

*arrays: inputs such as lists, arrays, data frames, or matrices

test_size: this is a float value whose value ranges between 0.0 and 1.0. it represents the proportion of our test size. its default value is none.

train_size: this is a float value whose value ranges between 0.0 and 1.0. it represents the proportion of our train size. its default value is none.

random_state: this parameter is used to control the shuffling applied to the data before applying the split. it acts as a seed.

shuffle: This parameter is used to shuffle the data before splitting. Its default value is true.

stratify: This parameter is used to split the data in a stratified fashion.

Read the dataset

In [3]:

```
df = pd.read_csv('/home/sumon/Documents/Datasets/sign_mnist_train.csv')
```

Get the locations

In [4]:

```
X = df.iloc[:, 1:785]
y = df.iloc[:, 0]
```

Split the dataset

In [5]:

```
seed = 7
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.33, random_state=seed)
```

Code to view first 5 records in the data set

In [6]:

```
df.head()
```

Out[6]:

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pixe
0	3	107	118	127	134	139	143	146	150	153	...	207	207	207	207	206	206	
1	6	155	157	156	156	156	157	156	158	158	...	69	149	128	87	94	163	
2	2	187	188	188	187	187	186	187	188	187	...	202	201	200	199	198	199	
3	2	211	211	212	212	211	210	211	210	210	...	235	234	233	231	230	226	
4	12	164	167	170	172	176	179	180	184	185	...	92	105	105	108	133	163	

5 rows × 785 columns

In [7]:

```
X.head()
```

Out[7]:

	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	pixel10	...	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pi
0	107	118	127	134	139	143	146	150	153	156	...	207	207	207	207	206	206	
1	155	157	156	156	156	157	156	158	158	157	...	69	149	128	87	94	163	
2	187	188	188	187	187	186	187	188	187	186	...	202	201	200	199	198	199	
3	211	211	212	212	211	210	211	210	210	211	...	235	234	233	231	230	226	
4	164	167	170	172	176	179	180	184	185	186	...	92	105	105	108	133	163	

5 rows × 784 columns

Training the XGBoost model

In [8]:

```
model = XGBClassifier()
model.fit(X_train, y_train)
```

Out[8]:

```
XGBClassifier(base_score=0.5, booster='gbtree', callbacks=None,
              colsample_bylevel=1, colsample_bynode=1, colsample_bytree=1,
              early_stopping_rounds=None, enable_categorical=False,
              eval_metric=None, feature_types=None, gamma=0, gpu_id=-1,
              grow_policy='depthwise', importance_type=None,
              interaction_constraints='', learning_rate=0.300000012,
              max_bin=256, max_cat_threshold=64, max_cat_to_onehot=4,
              max_delta_step=0, max_depth=6, max_leaves=0, min_child_weight=1,
              missing=nan, monotone_constraints='()', n_estimators=100,
              n_jobs=0, num_parallel_tree=1, objective='multi:softprob',
              predictor='auto', ...)
```

In [9]:

```
print(model)
```

```
XGBClassifier(base_score=0.5, booster='gbtree', callbacks=None,
              colsample_bylevel=1, colsample_bynode=1, colsample_bytree=1,
              early_stopping_rounds=None, enable_categorical=False,
              eval_metric=None, feature_types=None, gamma=0, gpu_id=-1,
              grow_policy='depthwise', importance_type=None,
              interaction_constraints='', learning_rate=0.300000012,
              max_bin=256, max_cat_threshold=64, max_cat_to_onehot=4,
              max_delta_step=0, max_depth=6, max_leaves=0, min_child_weight=1,
              missing=nan, monotone_constraints='()', n_estimators=100,
              n_jobs=0, num_parallel_tree=1, objective='multi:softprob',
              predictor='auto', ...)
```

Making predictions with the XGBoost model

Make predictions for test data

In [10]:

```
y_pred = model.predict(X_test)
predictions = [round(values) for values in y_pred]
```

Evaluate predictions

In [11]:

```
accuracy = accuracy_score(y_test, predictions)
print("Accuracy: %.2f%%" %(accuracy * 100))
```

```
Accuracy: 99.02%
```

Practical - 5

Aim: To implement Bagging algorithm for regression and prediction

Evaluate bagging ensemble for regression

Load required libraries

In [1]:

```
from numpy import mean
from numpy import std
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedKFold
from sklearn.ensemble import BaggingRegressor
```

We can use the make_regression() function to create a synthetic regression problem with 1,000 examples and 20 input features.

Define dataset

In [2]:

```
X, y = make_regression(n_samples=1000, n_features=20,
                       n_informative=15, noise=0.1, random_state=5)
```

Summarize the dataset

In [3]:

```
print(X.shape, y.shape)
```

```
(1000, 20) (1000,)
```

Define the model

In [4]:

```
model = BaggingRegressor()
```

We will evaluate the model using repeated k-fold cross-validation, with three repeats and 10 folds. We will report the mean absolute error (MAE) of the model across all repeats and folds. The scikit-learn library makes the MAE negative so that it is maximized instead of minimized. This means that larger negative MAE are better and a perfect model has a MAE of 0.

Evaluate the model

In [5]:

```
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
n_scores = cross_val_score(model, X, y,
                           scoring='neg_mean_absolute_error',
                           cv=cv, n_jobs=-1, error_score='raise')
```

Report performance

In [6]:

```
print('MAE: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

```
MAE: -101.427 (8.210)
```

We can also use the Bagging model as a final model and make predictions for regression.

First, the Bagging ensemble is fit on all available data, then the predict() function can be called to make predictions on new data.

Fit the model on the whole dataset

In [7]:

```
model.fit(X, y)
```

Out[7]:

```
BaggingRegressor()
```

Make a single prediction

In [8]:

```
row = [[0.88950817,-0.93540416,0.08392824,0.26438806,
        -0.52828711,-1.21102238,-0.4499934,1.47392391,
        -0.19737726,-0.22252503,0.02307668,0.26953276,
        0.03572757,-0.51606983,-0.39937452,1.8121736,
        -0.00775917,-0.02514283,-0.76089365,1.58692212]]
yhat = model.predict(row)
print('Prediction: %d' % yhat[0])
```

```
Prediction: -136
```

Practical - 6

Aim: To implement Bagging algorithm for classification and prediction

Explore bagging ensemble number of trees effect on performance

Load necessary libraries

In [1]:

```
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import BaggingClassifier
from matplotlib import pyplot
```

We can use the make_classification() function to create a synthetic binary classification problem with 1,000 examples and 20 input features.

Get the dataset

In [2]:

```
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20,
                             n_informative=15, n_redundant=5,
                             random_state=5)
  return X, y
```

Get a list of models to evaluate

In [3]:

```
def get_models():
    models = dict()
    # define number of trees to consider
    n_trees = [10, 50, 100, 500, 500, 1000, 5000]
    for n in n_trees:
        models[str(n)] = BaggingClassifier(n_estimators=n)
    return models
```

We can evaluate a Bagging algorithm on this dataset.

We will evaluate the model using repeated stratified k-fold cross-validation, with three repeats and 10 folds. We will report the mean and standard deviation of the accuracy of the model across all repeats and folds.

Evaluate a given model using cross-validation

In [4]:

```
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3,
                                 random_state=1)
    # evaluate the model and collect the results
    scores = cross_val_score(model, X, y, scoring='accuracy',
                             cv=cv, n_jobs=-1)
    return scores
```

Running the example reports the mean and standard deviation accuracy of the model.

Define dataset

In [5]:

```
X, y = get_dataset()
print(X.shape, y.shape)
```

```
(1000, 20) (1000,)
```

Get the models to evaluate

In [6]:

```
models = get_models()
```

Evaluate the models and store results

In [7]:

```
results, names = list(), list()
for name, model in models.items():
  scores = evaluate_model(model, X, y)  # evaluate the model
  results.append(scores)  # store the results
  names.append(name)
  # summarize the performance along the way
  print('>%s Accuracy: %.3f (%.3f)' % (name,mean(scores), std(scores)))
```

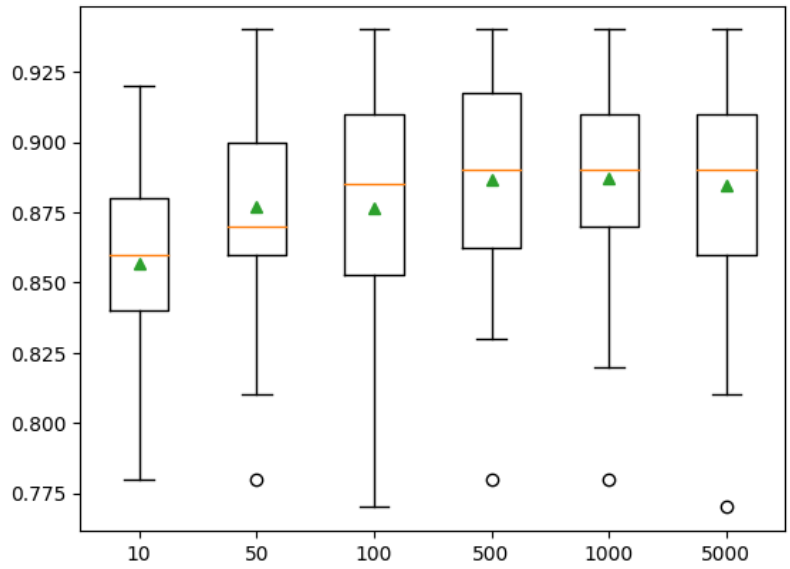
```
>10 Accuracy: 0.857 (0.035)
>50 Accuracy: 0.877 (0.037)
>100 Accuracy: 0.877 (0.040)
>500 Accuracy: 0.887 (0.037)
>1000 Accuracy: 0.887 (0.037)
>5000 Accuracy: 0.885 (0.037)
```

In this case, we can see that that performance improves on this dataset until about 100 trees and remains flat after that.

Plot model performance for comparison

In [8]:

```
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```



We can also use the Bagging model as a final model and make predictions for classification.

First, the Bagging ensemble is fit on all available data, then the predict() function can be called to make predictions on new data.

Fit the model on the whole dataset

In [9]:

```
model.fit(X, y)
```

Out[9]:

```
BaggingClassifier(n_estimators=5000)
```

Make a single prediction

In [10]:

```
row = [[-4.7705504,-1.88685058,-0.96057964,2.53850317,
        -6.5843005,3.45711663,-7.46225013,2.01338213,
        -0.45086384,-1.89314931,-2.90675203,-0.21214568,
        -0.9623956,3.93862591,0.06276375,0.33964269,
        4.0835676,1.31423977,-2.17983117,3.1047287]]
yhat = model.predict(row)
print('Predicted Class: %d' % yhat[0])
```

```
Predicted Class: 1
```

Practical - 7

Aim: To implement KNeighborsClassifier as the base algorithm in ensemble Bagging

Evaluate bagging with knn algorithm for classification

Load necessary libraries

In [1]:

```
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import BaggingClassifier
```

Define dataset

In [2]:

```
X, y = make_classification(n_samples=1000, n_features=20,
                           n_informative=15, n_redundant=5,
                           random_state=5)
```

Define the model

In [3]:

```
model = BaggingClassifier(base_estimator=KNeighborsClassifier())
```

Evaluate the model

In [4]:

```
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
n_scores = cross_val_score(model, X, y, scoring='accuracy',
                           cv=cv, n_jobs=-1, error_score='raise')
```

Report performance

In [5]:

```
print('Accuracy: %.3f (%.3f)' % (mean(n_scores), std(n_scores)))
```

```
Accuracy: 0.881 (0.038)
```

Practical - 8

Aim: To test different values of k to find the right balance of model variance to achieve good performance as a bagged ensemble.

The below example tests bagged KNN models with k values between 1 and 20.

Explore bagging ensemble k for knn effect on performance

Load necessary libraries

In [6]:

```
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import BaggingClassifier
from sklearn.neighbors import KNeighborsClassifier
from matplotlib import pyplot
```

Get the dataset

In [7]:

```
def get_dataset():
    X, y = make_classification(n_samples=1000, n_features=20,
                               n_informative=15, n_redundant=5,
                               random_state=5)
    return X, y
```

Get a list of models to evaluate

In [8]:

```
def get_models():
    models = dict()
    # evaluate k values from 1 to 20
    for i in range(1,21):
        # define the base model
        base = KNeighborsClassifier(n_neighbors=i)
        # define the ensemble model
        models[str(i)] = BaggingClassifier(base_estimator=base)
    return models
```

Evaluate a given model using cross-validation

In [9]:

```
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3,
                                 random_state=1)
    # evaluate the model and collect the results
    scores = cross_val_score(model, X, y, scoring='accuracy',
                             cv=cv, n_jobs=-1)
    return scores
```

Define dataset

In [10]:

```
X, y = get_dataset()
```

Get the models to evaluate

In [11]:

```
models = get_models()
```

Evaluate the models and store results

In [12]:

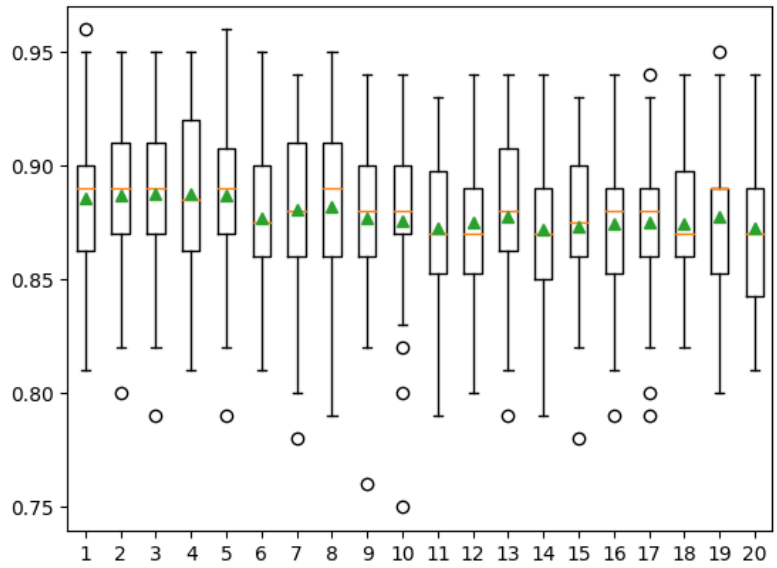
```
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
```

```
>1 0.886 (0.036)
>2 0.887 (0.033)
>3 0.888 (0.039)
>4 0.888 (0.036)
>5 0.887 (0.036)
>6 0.877 (0.035)
>7 0.881 (0.036)
>8 0.882 (0.036)
>9 0.877 (0.038)
>10 0.876 (0.040)
>11 0.873 (0.033)
>12 0.875 (0.034)
>13 0.878 (0.036)
>14 0.872 (0.036)
>15 0.874 (0.033)
>16 0.875 (0.034)
>17 0.875 (0.034)
>18 0.875 (0.032)
>19 0.878 (0.038)
>20 0.873 (0.034)
```

Plot model performance for comparison

In [13]:

```
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```



A box and whisker plot is created for the distribution of accuracy scores for each k value.

We see a general trend of increasing accuracy with sample size in the beginning, then a modest decrease in performance as the variance of the individual KNN models used in the ensemble is increased with larger k values.

Practical - 9

Aim: To evaluate the accuracy of bagging classifier with varying sample size

Explore bagging ensemble number of samples effect on performance

Load necessary libraries

In [1]:

```
from numpy import mean
from numpy import std
from numpy import arange
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import BaggingClassifier
from matplotlib import pyplot
```

The size of the bootstrap sample can be varied.

The default is to create a bootstrap sample that has the same number of examples as the original dataset. Using a smaller dataset can increase the variance of the resulting decision trees and could result in better overall performance.

The number of samples used to fit each decision tree is set via the "max_samples" argument.

The code below explores different sized samples as a ratio of the original dataset from 10 percent to 100 percent (the default).

Get the dataset

In [2]:

```
def get_dataset():
    X, y = make_classification(n_samples=1000, n_features=20,
                               n_informative=15, n_redundant=5,
                               random_state=5)
    return X, y
```

Get a list of models to evaluate

In [3]:

```
def get_models():
    models = dict()
    # explore ratios from 10% to 100% in 10% increments
    for i in arange(0.1, 1.1, 0.1):
        key = '%.1f' % i
        models[key] = BaggingClassifier(max_samples=i)
    return models
```

Evaluate a given model using cross-validation

In [4]:

```
def evaluate_model(model, X, y):
    # define the evaluation procedure
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3,
                                 random_state=1)
    # evaluate the model and collect the results
    scores = cross_val_score(model, X, y, scoring='accuracy',
                             cv=cv, n_jobs=-1)
    return scores
```

Define dataset

In [5]:

```
X, y = get_dataset()
```

Get the models to evaluate

In [6]:

```
models = get_models()
```

Evaluate the models and store results

In [7]:

```
results, names = list(), list()
for name, model in models.items():
    # evaluate the model
    scores = evaluate_model(model, X, y)
    # store the results
    results.append(scores)
    names.append(name)
    # summarize the performance along the way
    print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
```

```
>0.1 0.797 (0.035)
>0.2 0.834 (0.042)
>0.3 0.844 (0.039)
>0.4 0.849 (0.044)
>0.5 0.851 (0.036)
>0.6 0.852 (0.037)
>0.7 0.853 (0.033)
>0.8 0.859 (0.041)
>0.9 0.859 (0.036)
>1.0 0.863 (0.041)
```

In this case, the results suggest that performance generally improves with an increase in the sample size, highlighting that the default of 100 percent the size of the training dataset is sensible.

A box and whisker plot is created for the distribution of accuracy scores for each sample size.

We see a general trend of increasing accuracy with sample size.

Plot model performance for comparison

In [8]:

```
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```