CURNEW MEDTECH INNOVATIONS PRIVATE LIMITED

SD03Q08

-Kishore G (1832029)

Problem Statement:

To understand and find the suitable ML model for the given Social_Network_Ads.csv dataset, and to retrieve the given output plot.

Abstract:

The given problem deals with the creation of a suitable classification model for the given social network advertisement. The tool used here is Python. The model here is implemented by using Random forest algorithm. For the given dataset, we first explore the data and analyse each and every variable, then split data into train and test sets, define a Random Forest classifier from scratch and achieve the final output plot.

About the Dataset:

The dataset contains some information about all of our users in the social network, including their User ID, Gender, Age, and Estimated Salary. The last column of the dataset is a vector of Booleans describing whether or not each individual ended up clicking on the advertisement (0 = False, 1 = True).

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	User ID	Gender	Age	Estimated	Purchased										
2	15624510	Male	19	19000	0										
3	15810944	Male	35	20000	0										
4	15668575	Female	26	43000	0										
5	15603246	Female	27	57000	0										
6	15804002	Male	19	76000	0										
7	15728773	Male	27	58000	0										
8	15598044	Female	27	84000	0										
9	15694829	Female	32	150000	1										
10	15600575	Male	25	33000	0										
11	15727311	Female	35	65000	0										
12	15570769	Female	26	80000	0										
13	15606274	Female	26	52000	0										
14	15746139	Male	20	86000	0										
15	15704987	Male	32	18000	0										
16	15628972	Male	18	82000	0										
17	15697686	Male	29	80000	0										
18	15733883	Male	47	25000	1										
19	15617482	Male	45	26000	1										
20	15704583	Male	46	28000	1										

Exploratory Data Analysis:

First we read the dataset from the system. Then we extract X and Y variables. The X variable consists of Age and Estimated Salary, whereas the Y variable consists of the Purchase column, which states whether the customer has purchased or not**.** In order to understand the data better first we describe the data, which gives the summary statistics of each and every variable.

Code:

data.describe()

Output:

	User ID	Age	EstimatedSalary	Purchased
count	4.000000e+02	400.000000	400.000000	400.000000
mean	1.569154e+07	37.655000	69742.500000	0.357500
std	7.165832e+04	10.482877	34096.960282	0.479864
min	1.556669e+07	18.000000	15000.000000	0.000000
25%	1.562676e+07	29.750000	43000.000000	0.000000
50%	1.569434e+07	37.000000	70000.000000	0.000000
75%	1.575036e+07	46.000000	88000.000000	1.000000
max	1.581524e+07	60.000000	150000.000000	1.000000

Here the detailed summary for our data is given with count mean, std, min, max, etc. for all the variables in the dataset.

Code:

data.isnull().sum()

Output:

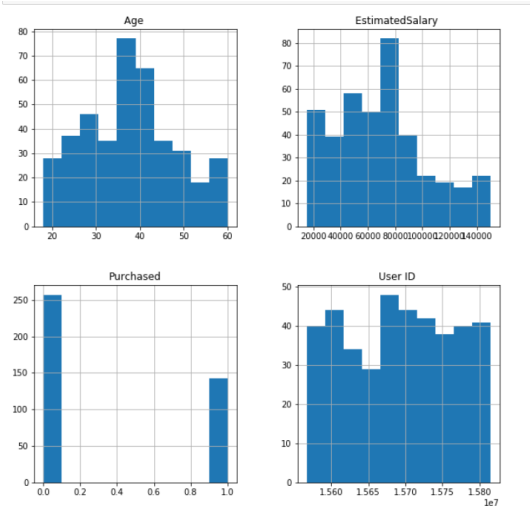
```
User ID             0
Gender              0
Age                 0
EstimatedSalary     0
Purchased           0
dtype: int64
```

Now we check for null values in the dataset. More the null values lesser the accuracy. Here we can see that there are no null values. So we need not pre-process the data.

Code:

data.hist(figsize=(10,10)) plt.show()

Output:



The histograms of each variable clearly shows their range and how they have influenced the data. Age falls between 20-60. Estimated Salary falls between 20k-1L. Purchased is the target variable which consists of 0s and 1s. User IDs are scattered throughout the data.

Finding the best ML model:

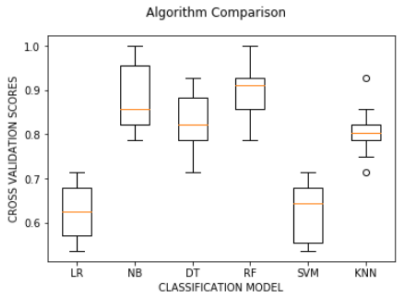
Code:

```
lr = LR(C = 0.2, max_iter = 1000) nb = NB()
dt = tree.DecisionTreeClassifier(criterion='entropy') rf = RFC(max_depth=5,
random_state=0)
svm = SVC(probability=True)
knn = KNN(n_neighbors = 5, metric = 'minkowski', p = 2) models = []
models.append(('LR', lr))
models.append(('NB', nb))
models.append(('DT', dt))
models.append(('RF', rf))
models.append(('SVM', svm))
models.append(('KNN', knn)) results =
[]
```

```
names = []

scoring = 'accuracy'

X_train,X_test,y_train,y_test = TTS(X, y, test_size = 0.3, random_state = 0)

for name, model in models:

    kfold = model_selection.KFold(n_splits = 10)

    cv_results = model_selection.cross_val_score(model, X_train, y_train, cv=kfold, scoring=scoring)

    results.append(cv_results)

    names.append(name)

    msg = "%s : %f (%f)" % (name, cv_results.mean(), cv_results.std())

    print(msg) fig =

plt.figure()

fig.suptitle('Algorithm Comparison') ax =

fig.add_subplot(111) plt.boxplot(results)

plt.xlabel("CLASSIFICATION MODEL")

plt.ylabel("CROSS VALIDATION SCORES")

ax.set_xticklabels(names) plt.show()
```

Output:

```
LR   : 0.625000 (0.066336)
NB   : 0.885714 (0.076265)
DT   : 0.832143 (0.065951)
RF   : 0.896429 (0.062780)
SVM  : 0.628571 (0.068139)
KNN  : 0.807143 (0.055787)
```



From the boxplot we can clearly infer that Random Forest is the best model

Here we have built a code and a boxplot that gives the accuracies of all the models, from which we found that Random forest is the best model that suits our dataset with an accuracy of 89%. So we will now build the random forest classifier from scratch.

Implementing the Random Forest Classifier:

Random forest is a flexible, easy to use machine learning algorithm that produces, even without hyper-parameter tuning, a great result most of the time. It is also one of the most used algorithms, because of its simplicity and diversity (it can be used for both classification and regression tasks).

Code:

```
def entropy(y):
    hist = np.bincount(y) ps = hist /
    len(y)
    return -np.sum([p * np.log2(p) for p in ps if p > 0]) class Node:
    def init (self, feature=None, threshold=None, left=None, right=None, *,
value=None):
        self.feature = feature self.threshold =
        threshold self.left = left self.right =
        right self.value = value
    def is_leaf_node(self):
        return self.value is not None class
DecisionTree:
```

```
    def init (self, min_samples_split=2, max_depth=100, n_feats=None):
        self.min_samples_split = min_samples_split self.max_depth =
        max_depth
        self.n_feats = n_feats self.root =
        None
    def fit(self, X, y):
        self.n_feats = X.shape[1] if not self.n_feats else min(self.n_feats,
X.shape[1])
        self.root = self._grow_tree(X, y) def predict(self,
    X):
        return np.array([self._traverse_tree(x, self.root) for x in
X])
    def _grow_tree(self, X, y, depth=0): n_samples,
        n_features = X.shape n_labels =
        len(np.unique(y))
        # stopping criteria
        if (depth >= self.max_depth or n_labels
                == 1
                or n_samples < self.min_samples_split): leaf_value =
            self._most_common_label(y) return Node(value=leaf_value)
        feat_idxs = np.random.choice(n_features, self.n_feats,
replace=False)
                    # greedily select the best split according to information
gain
        best_feat, best_thresh = self._best_criteria(X, y,
feat_idxs)
        # grow the children that result from the split left_idxs, right_idxs =
        self._split(X[:, best_feat],
best_thresh)
        left = self._grow_tree(X[left_idxs, :], y[left_idxs], depth+1)
        right = self._grow_tree(X[right_idxs, :], y[right_idxs], depth+1)
        return Node(best_feat, best_thresh, left, right) def _best_criteria(self, X,
    y, feat_idxs):
        best_gain = -1
```

```
            split_idx, split_thresh = None, None for feat_idx in
            feat_idxs:
                X_column = X[:, feat_idx] thresholds =
                np.unique(X_column) for threshold in
                thresholds:
                    gain = self._information_gain(y, X_column,
threshold)
                    if gain > best_gain: best_gain = gain
                        split_idx = feat_idx split_thresh =
                        threshold
            return split_idx, split_thresh
        def _information_gain(self, y, X_column, split_thresh): # parent loss
            parent_entropy = entropy(y) # generate
            split
            left_idxs, right_idxs = self._split(X_column, split_thresh) if len(left_idxs) == 0 or
            len(right_idxs) == 0:
                return 0
            # compute the weighted avg. of the loss for the children n = len(y)
            n_l, n_r = len(left_idxs), len(right_idxs)
            e_l, e_r = entropy(y[left_idxs]), entropy(y[right_idxs]) child_entropy = (n_l / n) *
            e_l + (n_r / n) * e_r
            # information gain is difference in loss before vs. after
split
            ig = parent_entropy - child_entropy return ig
        def _split(self, X_column, split_thresh):
            left_idxs = np.argwhere(X_column <= split_thresh).flatten() right_idxs =
            np.argwhere(X_column > split_thresh).flatten() return left_idxs, right_idxs
        def _traverse_tree(self, x, node): if
            node.is_leaf_node():
                return node.value
```

```
            if x[node.feature] <=  node.threshold: return self._traverse_tree(x,
                node.left)
            return self._traverse_tree(x, node.right) def
        _most_common_label(self, y):
            counter = Counter(y)
            most_common = counter.most_common(1)[0][0] return
            most_common
import numpy as np
from collections import Counter def
bootstrap_sample(X, y):
        n_samples = X.shape[0]
        idxs = np.random.choice(n_samples, n_samples, replace=True) return X[idxs], y[idxs]
def most_common_label(y): counter
        = Counter(y)
        most_common = counter.most_common(1)[0][0] return
        most_common
class RandomForest:
        def_init_(self, n_trees=10, min_samples_split=2, max_depth=100, n_feats=None):
            self.n_trees = n_trees self.min_samples_split =
            min_samples_split self.max_depth = max_depth
            self.n_feats = n_feats self.trees =
            []
        def fit(self, X, y): self.trees =
            []
            for _ in range(self.n_trees):
                tree =
DecisionTree(min_samples_split=self.min_samples_split,
                    max_depth=self.max_depth, n_feats=self.n_feats) X_samp, y_samp =
                bootstrap_sample(X, y) tree.fit(X_samp, y_samp)
                self.trees.append(tree)
```

```
    def predict(self, X):
        tree_preds = np.array([tree.predict(X) for tree in self.trees])
        tree_preds = np.swapaxes(tree_preds, 0, 1)
        y_pred = [most_common_label(tree_pred) for tree_pred in tree_preds]
        return np.array(y_pred) import
numpy as np
import pandas as pd
from sklearn import datasets
from sklearn.model_selection import train_test_split def accuracy(y_true,
y_pred):
    accuracy = np.sum(y_true == y_pred) / len(y_true) return accuracy
data = pd.read_csv(r'C:\Users\Selva Vignesh
M\Desktop\Social_Network_Ads.csv')
X = data.iloc[:, [2, 3]].values y = data.iloc[:,
4].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=123)
clf = RandomForest(n_trees=3, max_depth=10) clf.fit(X_train,
y_train)
y_pred = clf.predict(X_test) acc =
accuracy(y_test, y_pred) print ("Accuracy:",
acc)
from sklearn.metrics import confusion_matrix,classification_report
print('Classification Report \n', classification_report(y_test, y_pred))
from sklearn.metrics import confusion_matrix cm =
confusion_matrix(y_test, y_pred) print(cm)
```

Output:

```
Accuracy: 0.8625
Classification Report
              precision    recall  f1-score   support

           0       0.91      0.86      0.89        50
           1       0.79      0.87      0.83        30

    accuracy                           0.86        80
   macro avg       0.85      0.86      0.86        80
weighted avg       0.87      0.86      0.86        80

[[43  7]
 [ 4 26]]
```

The accuracy we achieved from the random forest model which is built from scratch is 86% which is a good accuracy. Using built-in functions will surely improve the accuracy of the model. Now comes the classification report which displays the precision, recall, F1 and support scores for the model.

- Precision Score for class 0 (negative) is 0.91 and for class 1 (positive) is 0.84, indicating the preciseness of the model which is so accurate.
- Recall value for class 0 is 0.76 and class 1 is 0.87, which describes the amount up – to which the model can predict the output.
- As the precision and recall values are not similar, there are dominance in classes.

From the confusion matrix we can see that there are 43 true positives, 7 false positives, 4 false negatives and 26 true negative values.

Built-in Random Forest model:

In order to obtain the given output plot we need to build once again the random forest model using built-in functions as scratch model did not achieve the exact plot.

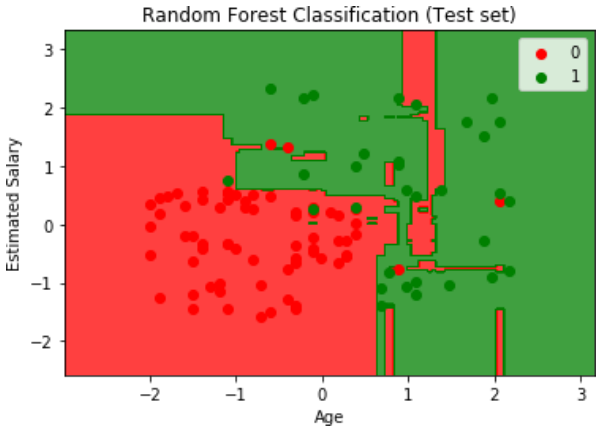
Code:

```
from sklearn.ensemble import RandomForestClassifier
classifier = RandomForestClassifier(n_estimators = 10, criterion = 'entropy', random_state = 0)
classifier.fit(X_train, y_train) y_pred =
classifier.predict(X_test)
```

Model Visualization:

```
from matplotlib.colors import ListedColormap from matplotlib

import pyplot as plt

X_set, y_set = X_test, y_test

X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step =
0.01),

                                    np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1,
step = 0.01))

plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(),
X2.ravel()]).T).reshape(X1.shape),

                        alpha = 0.75, cmap = ListedColormap(('red', 'green'))) plt.xlim(X1.min(),

X1.max())

plt.ylim(X2.min(), X2.max())

for i, j in enumerate(np.unique(y_set)): plt.scatter(X_set[y_set == j, 0], X_set[y_set ==

    j, 1],

                        c = ListedColormap(('red', 'green'))(i), label = j) plt.title('Random Forest

Classification (Test set)') plt.xlabel('Age')

plt.ylabel('Estimated Salary') plt.legend()

plt.show()
```

Output:



Conclusion:

But while there is always room for improvement, we can be satisfied with this model as our final product. Our accuracy is high, but not so high that we need to be suspicious of any over fitting. We can safely say that an increase in both Age and Estimated Salary will lead to a

higher probability of clicking the advertisement. As new users sign-up for the website, we can use this model to quickly determine whether or not to expose them to this particular ad or choose another that is more relevant to their profile.