University of Essex

CE802 MACHINE LEARNING

Assignment 1: Design and Application of a Machine Learning System for a Practical Problem

Name: Ahmad Raza
Registration number: 2101194
PRID: RAZAA69707
Masters in Artificial Intelligence
2021-2022

Word count: 1622
(Excluding the title page, diagrams, tables, references,headings)

Comparative Study

In the part 2 of assignment, we have implemented supervised learning classification models shortlisted in the pilot-study and compared their results based on the accuracy, precision, recall and F1. These models include SVM, KNN, Random Forest Classifier and Decision Tree Classifier. Among these every model has its advantages as well as disadvantages and they give different results on different datasets.

Steps Involved in Experimenting the different Implementations of Models Explained:

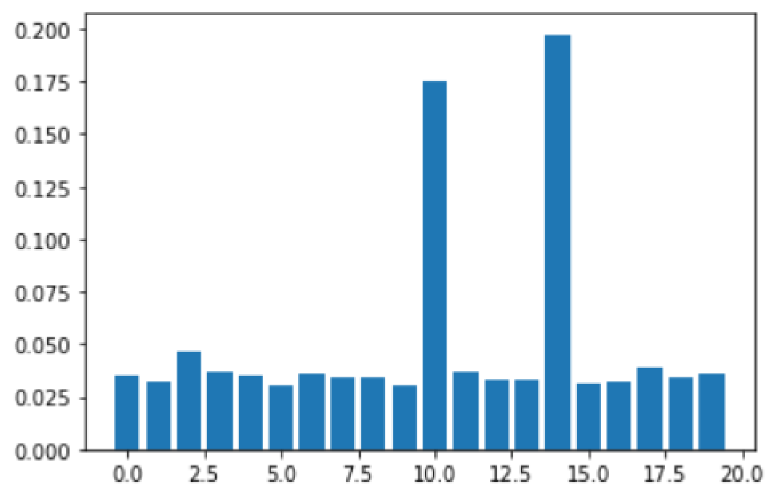
After importing the necessary **libraries** and **reading** the data from the .csv files provided, and separating our input columns and target column, we perform some feature engineering.

In first step, co-relation is checked between the features and highly co-related features are removed. But surprisingly, no features in given data are co-related to each other more than 70 percent.

```
correlation = FX1.corr().abs()
upper_triangle = correlation.where(np.triu(np.ones(correlation.shape),k=1).astype(bool))
#dropping columns having correlation greater than 90 percent
dropping = [column for column in upper_triangle.columns if any(upper_triangle[column] > 0.70)]
print(dropping)
```

```
[]
```

In final step, we look for the features importance in every classifier and graph them.



After feature engineering, we check for any null values. It seems that the F21 column contains null values.

```
X1=train_data.drop(columns=['Class','F21'])
Y1=train_data['Class']
```

1. Imputation Methods:

To tackle the null values in F21 there are multiple imputation methods that can be used but due to the shortage of time, we have only experimented on following three methods:

- Drop the F21 column
- Impute the mean value using SimpleImputer
- Impute the mean value using IterativeImputer

2. Splitting the data:

Data is split into test and train data where 80 percent of the data is allocated to training and rest of the 20 percent is allocated to the testing.

3. Stratification:

While splitting the data, random stratification is performed to obtain a sample population that best represents the entire population being studied. [1]

```
X, X_test, Y, Y_test = train_test_split(X1, Y1, train_size=0.80, random_state=1, stratify=Y1)
```

4. Two Different Approaches:

After the data splitting, we use two different approaches to implement the algorithms. In the first approach following steps are performed:

i. Implementation without hyper-parameter tuning, cross-validation, pipelining, etc:

a. **Scaling:** Once data is split, we perform scaling on the data. For that purpose, we use the StandardScalar(). We fit the model on train data and then transform on both train and test data.

```
scalar=StandardScaler()
X=scalar.fit_transform(X)
X_test=scalar.transform(X_test)
```

b. **Fitting the model:** After scaling is performed, we train the desired model on the training data.

```
clf=KNeighborsClassifier()
clf.fit(X,Y)
```

c. **Prediction:** Once model is trained, we predict the values for the test data using the trained model.

```
y_pred = clf.predict(X_test)
```

d. **Analyzing:** Once the values are predicted, based on the trained model, we analyze the results based on different metrics such as test accuracy, precision, recall and F1 score.

```
print('Confusion Matrix : \n' + str(confusion_matrix(Y_test,y_pred)))
print('Accuracy Score : ' + str(accuracy_score(Y_test,y_pred)))
print('F1 Score : ' + str(f1_score(Y_test,y_pred)))
print('Precision Score : ' + str(precision_score(Y_test,y_pred)))
print('Recall Score : ' + str(recall_score(Y_test,y_pred)))
```

ii. Implementation using hyper-parameter tuning, cross-validation, pipelining:

While performing algorithms using the first approach, we are facing a few problems.

- We are able to get the accuracy and other evaluation metrics but there is no way to know if our model is under-fitted or over-fitted and how is it going to perform on the actual real-world un-seen data. To tackle this issue we are going to perform **K-Fold cross-validation.**
- Another issue is that we are not tuning the hyper-parameters therefore, the algorithm performance cannot be considered ideal as the algorithm is running on the default values. This issue can be solved by **tuning the hyper-parameters by using GridSearchCV.**
- Another issue that we face is that all the steps in the learning process are to be performed manually, therefore we need to automate those steps so that every step is performed in the defined sequence. This automation can be done using the **pipeline.**

For second approach we will perform following steps:

a. Defining the pipeline: We will define the steps in our pipeline for example, Imputation->Scaling->Model.

```
mean= SimpleImputer(strategy='mean', fill_value='Missing')
scalar=StandardScaler()
knn = KNeighborsClassifier()
pipe = Pipeline([('mean', mean), ('scalar',scalar) , ('knn', knn)])
```

b. Defining the parameters dictionary: We add all the parameters that we want to tune and their possible values inside this dictionary.

```
param_grid = {'knn__leaf_size':list(range(1,45)), 'knn__n_neighbors':list(range(1,100)), 'knn__p': [1,2] , 'knn__weights': ['uni
```

c. **Performing Grid Search:** After defining our pipeline and parameter dictionary, we pass both these to the GridSearchCV that search all the possible combinations and return us the pair of hyper-parameters on which our model is performing the best.

```
search = GridSearchCV(pipe, param_grid, n_jobs=-1)
```

d. **Cross Validation:** Cross validation is performed by using cross_validate. By performing, cross-validation we get the accuracy for every fold and then calculate the mean accuracy so that ocne we have our test accuracy, we can compare it with this average training accuracy to see if our model is under-fitting, over-fitting or performing well.

```
scores = cross_validate(search, X, Y, scoring=['accuracy'], cv=cv, return_estimator=True)
print('Average accuracy', np.mean(scores['test_accuracy']))
```

e. **Training**: Now, we train our model on the best-parameters obtained by using GridSearch.

f. **Pre-processing of test data:** After the training, we pre-process the test data based on the pipeline defined for the pre-processing of test data.

```
pipes = Pipeline([('mean', mean)])
pipes.fit_transform(X_test)
```

g. **Prediction:** Once model is trained, we predict the values for the test data on the trained model.

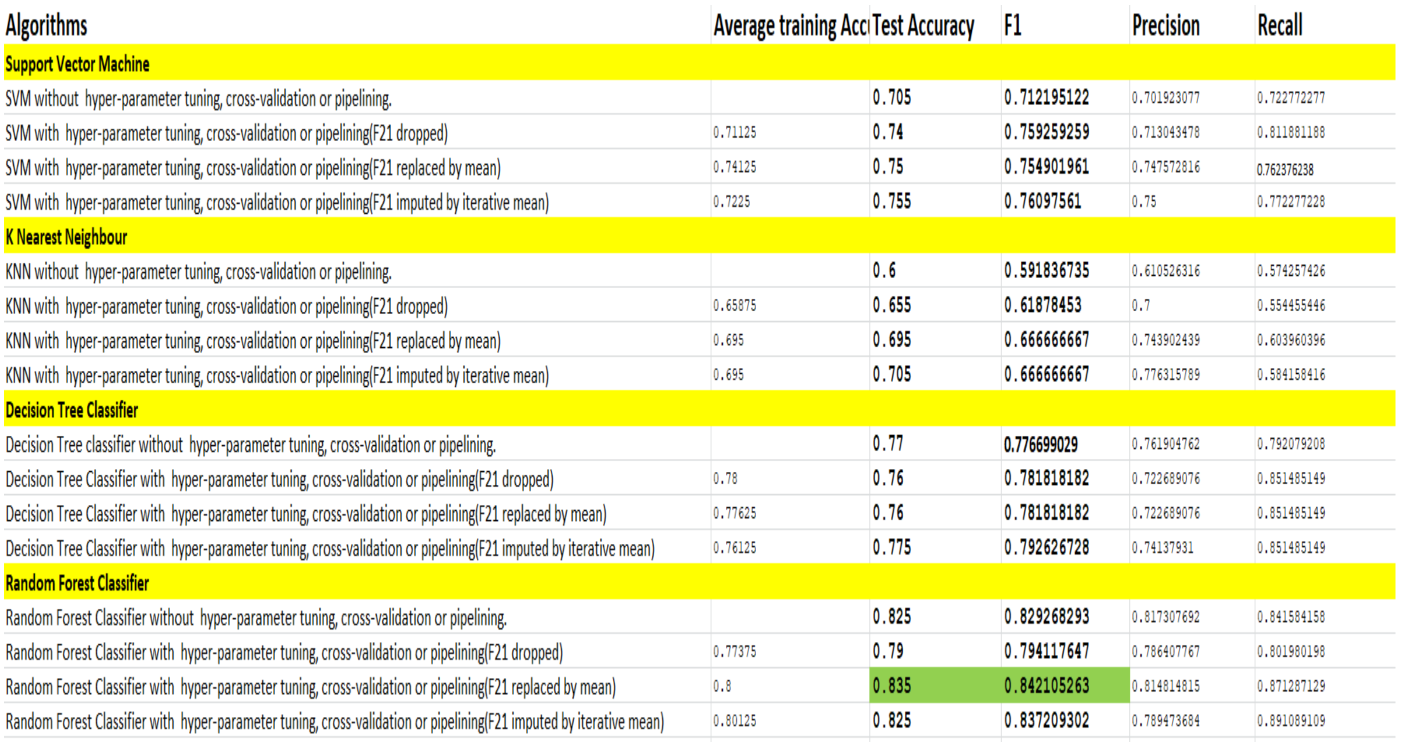
```
y_pred = clf.predict(X_test)
```

h. **Analysis:** After the predictions, we analyze the model based on different metrics such as training accuracy, test accuracy, F1-score, etc.

```
print('Confusion Matrix : \n' + str(confusion_matrix(Y_test,y_pred)))
print('Accuracy Score : ' + str(accuracy_score(Y_test,y_pred)))
print('F1 Score : ' + str(f1_score(Y_test,y_pred)))
print('Precision Score : ' + str(precision_score(Y_test,y_pred)))
print('Recall Score : ' + str(recall_score(Y_test,y_pred)))
```

Results Obtained by Experimenting different approaches:

Note: For better understanding of the results and convenience, I have put all the results in the same table instead of using multiple tables.



Algorithms	Average training Acci	Test Accuracy	F1	Precision	Recall
Support Vector Machine					
SVM without hyper-parameter tuning, cross-validation or pipelining.		0.705	0.712195122	0.701923077	0.722772277
SVM with hyper-parameter tuning, cross-validation or pipelining(F21 dropped)	0.71125	0.74	0.759259259	0.713043478	0.811881188
SVM with hyper-parameter tuning, cross-validation or pipelining(F21 replaced by mean)	0.74125	0.75	0.754901961	0.747572816	0.762376238
SVM with hyper-parameter tuning, cross-validation or pipelining(F21 imputed by iterative mean)	0.7225	0.755	0.76097561	0.75	0.772277228
K Nearest Neighbour					
KNN without hyper-parameter tuning, cross-validation or pipelining.		0.6	0.591836735	0.610526316	0.574257426
KNN with hyper-parameter tuning, cross-validation or pipelining(F21 dropped)	0.65875	0.655	0.61878453	0.7	0.554455446
KNN with hyper-parameter tuning, cross-validation or pipelining(F21 replaced by mean)	0.695	0.695	0.666666667	0.743902439	0.603960396
KNN with hyper-parameter tuning, cross-validation or pipelining(F21 imputed by iterative mean)	0.695	0.705	0.666666667	0.776315789	0.584158416
Decision Tree Classifier					
Decision Tree classifier without hyper-parameter tuning, cross-validation or pipelining.		0.77	0.776699029	0.761904762	0.792079208
Decision Tree Classifier with hyper-parameter tuning, cross-validation or pipelining(F21 dropped)	0.78	0.76	0.781818182	0.722689076	0.851485149
Decision Tree Classifier with hyper-parameter tuning, cross-validation or pipelining(F21 replaced by mean)	0.77625	0.76	0.781818182	0.722689076	0.851485149
Decision Tree Classifier with hyper-parameter tuning, cross-validation or pipelining(F21 imputed by iterative mean)	0.76125	0.775	0.792626728	0.74137931	0.851485149
Random Forest Classifier					
Random Forest Classifier without hyper-parameter tuning, cross-validation or pipelining.		0.825	0.829268293	0.817307692	0.841584158
Random Forest Classifier with hyper-parameter tuning, cross-validation or pipelining(F21 dropped)	0.77375	0.79	0.794117647	0.786407767	0.801980198
Random Forest Classifier with hyper-parameter tuning, cross-validation or pipelining(F21 replaced by mean)	0.8	0.835	0.842105263	0.814814815	0.871287129
Random Forest Classifier with hyper-parameter tuning, cross-validation or pipelining(F21 imputed by iterative mean)	0.80125	0.825	0.837209302	0.789473684	0.891089109

- On observing the above results it can be seen that for SVM and Random Forest Classifier, test accuracies are a lit bit higher than the cross-validation accuracies but the difference is varying between 1 to 3 percent only so I would assume that it's not necessary that our model is under-fitting because it's a small dataset so the difference is negligible.
- For KNN, both test and cross-validation accuracies are same with only a minor difference that is less than 1 percent so our model is trained well.
- For Decision Trees Classifier, our cross-validation accuracies are a little bit higher than the test accuracies, but only by 1 to 2 percent, therefore this difference is very negligible.
- Moreover, we can see that Random Forest Classifier is the best performing classifier in our case with highest **accuracy of 83.5** percent and **F1 score of 84 percent** while kNN is the worst performing algorithm.
- Another important observation is that for all algorithms hyper-parameter tuning improves the test accuracy and F1 scores.

- SVM, KNN and decision tree algorithms perform best when F21 missing values are replaced using Iterative Mean.
- While for Random Forest Classifier, it performs best when F21 missing values are replaced by Simple Mean.

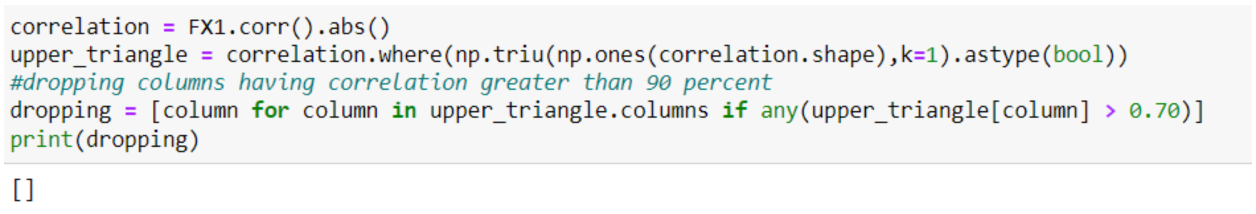
Additional Comparative Study

In the part 3 of assignment, we have implemented supervised learning regression models and compared their results based on the co-efficient of determination and root mean square error. These models include SVR, Random Forest Regression and Linear Regression. Among these every model has its advantages as well as disadvantages and they give different results on different datasets.

Steps Involved in Experimenting the different Implementations of Models Explained:

After importing the necessary **libraries** and **reading** the data from the .csv files provided, and separating our input columns and target column, we perform some feature engineering.

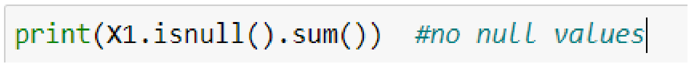
Co-relation is checked between the features and highly co-related features are removed. But surprisingly, no features in given data are co-related to each other more than 70 percent.



```
correlation = FX1.corr().abs()
upper_triangle = correlation.where(np.triu(np.ones(correlation.shape),k=1).astype(bool))
#dropping columns having correlation greater than 90 percent
dropping = [column for column in upper_triangle.columns if any(upper_triangle[column] > 0.70)]
print(dropping)
```

```
[]
```

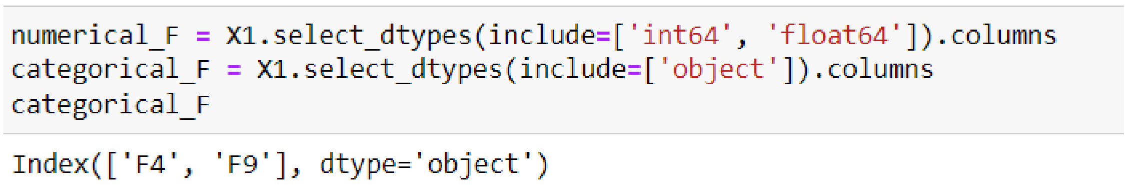
After feature engineering, we check for any null values. No column contains null values.



```
print(X1.isnull().sum())  #no null values
```

5. Imputation Methods:

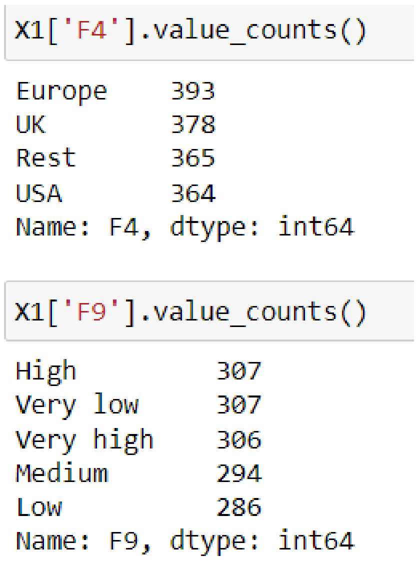
Once null values are checked, we further check for the data types of columns.



```
numerical_F = X1.select_dtypes(include=['int64', 'float64']).columns
categorical_F = X1.select_dtypes(include=['object']).columns
categorical_F
```

```
Index(['F4', 'F9'], dtype='object')
```

It seems that F4 and F9 contains the categorical values. To deal with this we will be checking how many types of values these columns contain.



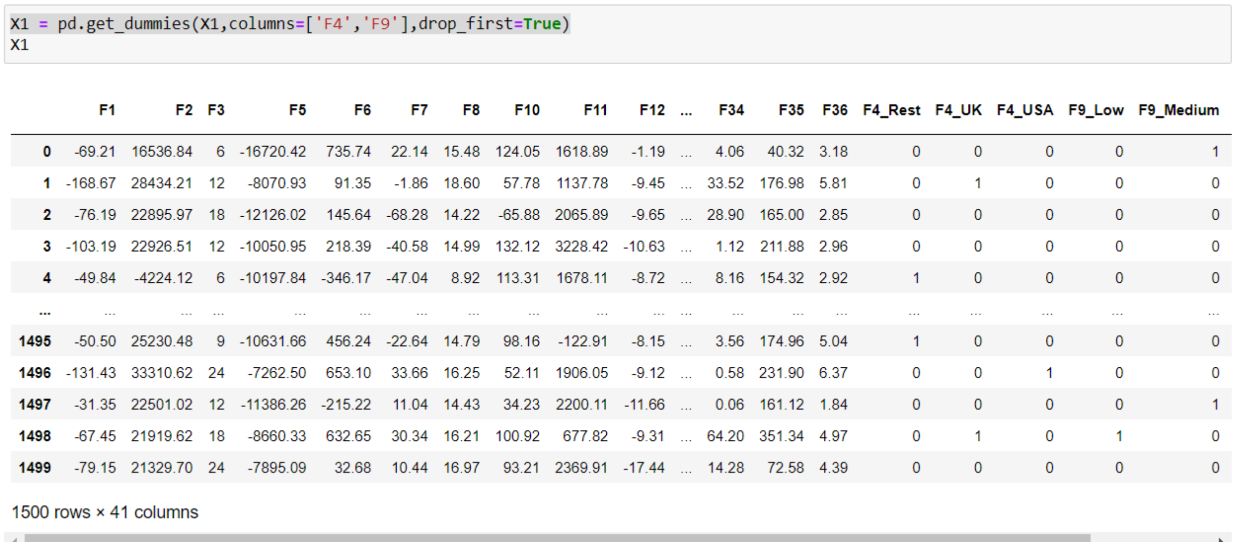
```
X1['F4'].value_counts()
```

```
Europe    393
UK        378
Rest      365
USA       364
Name: F4, dtype: int64
```

```
X1['F9'].value_counts()
```

```
High         307
Very low     307
Very high    306
Medium       294
Low          286
Name: F9, dtype: int64
```

As both columns contain 4 and 5 values respectively therefore we can use one-hot encoding. Details of one hot encoding can be read from the pre-processing lab of CE802MachineLearning Module. Pandas provide a get_dummies function that makes this job easy for us. Now, our data pre-processing is complete and we need to move forward with data splitting.



```
X1 = pd.get_dummies(X1,columns=['F4','F9'],drop_first=True)
X1
```

	F1	F2	F3	F5	F6	F7	F8	F10	F11	F12	...	F34	F35	F36	F4_Rest	F4_UK	F4_USA	F9_Low	F9_Medium
0	-69.21	16536.84	6	-16720.42	735.74	22.14	15.48	124.05	1618.89	-1.19	...	4.06	40.32	3.18	0	0	0	0	1
1	-168.67	28434.21	12	-8070.93	91.35	-1.86	18.60	57.78	1137.78	-9.45	...	33.52	176.98	5.81	0	1	0	0	0
2	-76.19	22895.97	18	-12126.02	145.64	-68.28	14.22	-65.88	2065.89	-9.65	...	28.90	165.00	2.85	0	0	0	0	0
3	-103.19	22926.51	12	-10050.95	218.39	-40.58	14.99	132.12	3228.42	-10.63	...	1.12	211.88	2.96	0	0	0	0	0
4	-49.84	-4224.12	6	-10197.84	-346.17	-47.04	8.92	113.31	1678.11	-8.72	...	8.16	154.32	2.92	1	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1495	-50.50	25230.48	9	-10631.66	456.24	-22.64	14.79	98.16	-122.91	-8.15	...	3.56	174.96	5.04	1	0	0	0	0
1496	-131.43	33310.62	24	-7262.50	653.10	33.66	16.25	52.11	1906.05	-9.12	...	0.58	231.90	6.37	0	0	1	0	0
1497	-31.35	22501.02	12	-11386.26	-215.22	11.04	14.43	34.23	2200.11	-11.66	...	0.06	161.12	1.84	0	0	0	0	1
1498	-67.45	21919.62	18	-8660.33	632.65	30.34	16.21	100.92	677.82	-9.31	...	64.20	351.34	4.97	0	1	0	1	0
1499	-79.15	21329.70	24	-7895.09	32.68	10.44	16.97	93.21	2369.91	-17.44	...	14.28	72.58	4.39	0	0	0	0	0

1500 rows × 41 columns

6. Splitting the data:

Data is split into test and train data where 80 percent of the data is allocated to training and rest of the 20 percent is allocated to the testing.

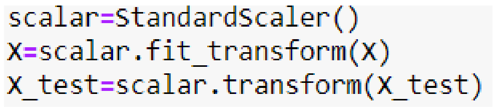
```
X, X_test, Y, Y_test = train_test_split(X1, Y1, train_size=0.8, random_state=1)
```

7. Two Different Approaches:

After the data splitting, we use two different approaches to implement the algorithms. In the first approach following steps are performed:

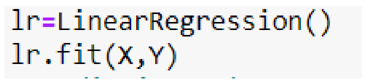
iii. Implementation without hyper-parameter tuning, cross-validation, pipelining, etc:

e. **Scaling:** Once data is split, we perform scaling on the data. For that purpose, we use the StandardScalar(). We fit the model on train data and then transform on both train and test data.



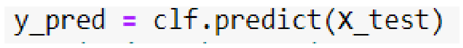
```
scalar=StandardScaler()
X=scalar.fit_transform(X)
X_test=scalar.transform(X_test)
```

f. **Fitting the model:** After scaling is performed, we train the desired model on the training data.



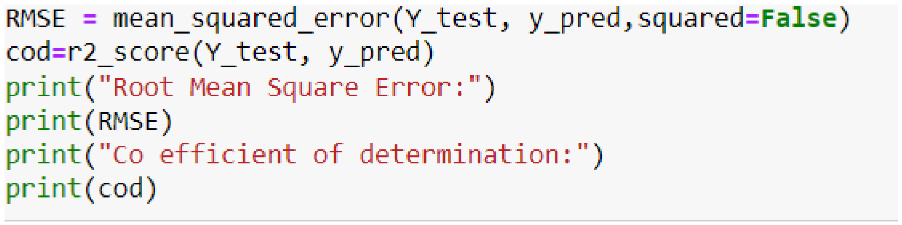
```
lr=LinearRegression()
lr.fit(X,Y)
```

g. **Prediction:** Once model is trained, we predict the values for the test data using the trained model.



```
y_pred = clf.predict(X_test)
```

h. **Analyzing:** Once the values are predicted, based on the trained model, we analyze the results based on different metrics such as rmse and r2 scores.



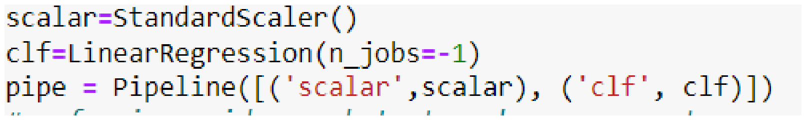
```
RMSE = mean_squared_error(Y_test, y_pred,squared=False)
cod=r2_score(Y_test, y_pred)
print("Root Mean Square Error:")
print(RMSE)
print("Co efficient of determination:")
print(cod)
```

iv. Implementation using hyper-parameter tuning, cross-validation, pipelining:

While performing algorithms using the first approach, we are facing a few problems that are already discussed while performing classification tasks.

For second approach we will perform following steps:

i. **Defining the pipeline:** We will define the steps in our pipeline for example, Scaling->Model.



```
scalar=StandardScaler()
clf=LinearRegression(n_jobs=-1)
pipe = Pipeline([('scalar',scalar), ('clf', clf)])
```

j. **Defining the parameters dictionary:** We add all the parameters that we want to tune and their possible values inside this dictionary.

```
param_grid = {'clf__fit_intercept':[True, False],'clf__normalize':[True,False],'clf__copy_X' :[True,False],'clf__positive' :[Tru
```

k. **Performing Grid Search:** After defining our pipeline and parameter dictionary, we pass both these to the GridSearchCV that search all the possible combinations and return us the pair of hyper-parameters on which our model is performing the best.

```
search = GridSearchCV(pipe, param_grid, n_jobs=-1)
```

l. **Cross Validation:** Cross validation is performed by using cross_validate. By performing, cross-validation we get the rmse for every fold and then calculate the mean rmse so that once we have our test rmse, we can compare it with this cross validation rmse to see if our model is under-fitting, over-fitting or performing well.

```
scores = cross_validate(search, X, Y, scoring=['neg_root_mean_squared_error'], cv=cv, return_estimator=True)
print('croos validation rmse', np.mean(scores['test_neg_root_mean_squared_error']))
```

m. **Training**: Now, we train our model on the best-parameters obtained by using GridSearch.

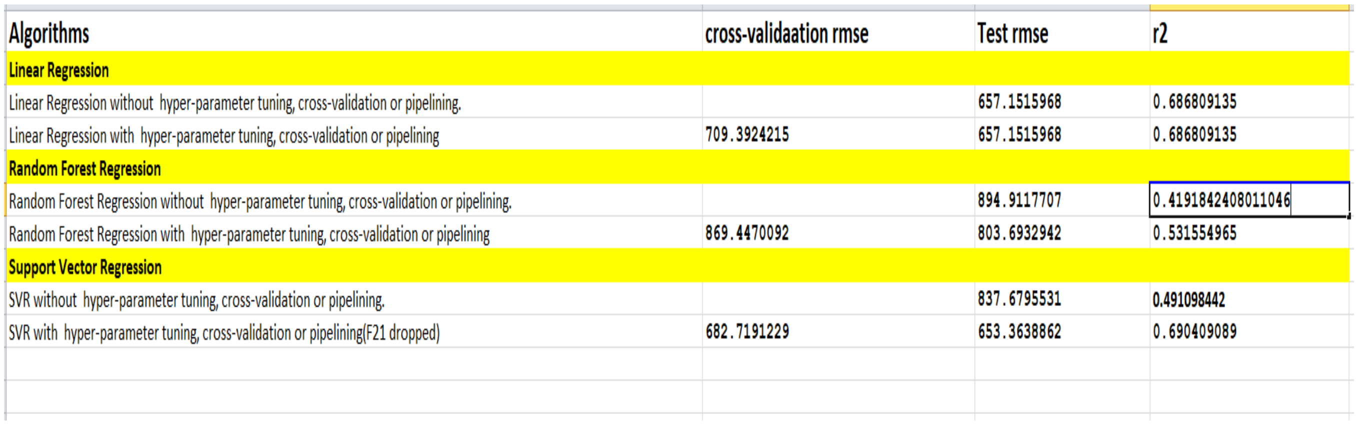
n. **Prediction:** Once model is trained, we predict the values for the test data on the trained model.

```
y_pred = clf.predict(X_test)
```

o. **Analysis:** After the predictions, we analyze the model based on different metrics such as cross validation rmse, rmse, and co efficient of variance.

```
#Analyzing the results
RMSE = mean_squared_error(Y_test, y_pred,squared=False)
cod=r2_score(Y_test, y_pred)
print("Root Mean Square Error:")
print(RMSE)
print("Co efficient of determination:")
print(cod)
```

Results Obtained by Experimenting different approaches:



Algorithms	cross-validaation rmse	Test rmse	r2
Linear Regression			
Linear Regression without hyper-parameter tuning, cross-validation or pipelining.		657.1515968	0.686809135
Linear Regression with hyper-parameter tuning, cross-validation or pipelining	709.3924215	657.1515968	0.686809135
Random Forest Regression			
Random Forest Regression without hyper-parameter tuning, cross-validation or pipelining.		894.9117707	0.4191842408011046
Random Forest Regression with hyper-parameter tuning, cross-validation or pipelining	869.4470092	803.6932942	0.531554965
Support Vector Regression			
SVR without hyper-parameter tuning, cross-validation or pipelining.		837.6795531	0.491098442
SVR with hyper-parameter tuning, cross-validation or pipelining(F21 dropped)	682.7191229	653.3638862	0.690409089

- From the above table, it can been that random forest regression and SVR root mean square errors are decreased significantly and their co-efficient of determination are also improved after tuning the hyper-parameters.
- While for linear regression values before and after tuning remain same because there are no significant hyper parameters for linear regression so gridsearchcv is not much helpful.
- SVR is the best performing algorithm among all when hyper parameters are tuned well as it can be seen that after tuning its rmse value is least among other two algorithms and also it has the highest co efficient of variance.
- Random forest regressor give the worst results during our experiments. Even after hyper parameter tuning it has the highest rmse.
- For all the algorithms, cross validation rmse are a little bit higher than the test root mean square errors but it's a negligible difference so it can be ignored.

References:

[1] https://www.investopedia.com/terms/stratified_random_sampling.asp