Assignment 1 - Code Smell detector

Juan Pablo Lozano Gómez

Part 1. System Design, Training scripts execution and Hyper parameter tuning

To train the model I decided to implement a system that could do several things.

1. Load a data set
2. Clean the data set and fill the empty values when needed (df.isnull().sum().index > 0) and convert the dependent or Y data into binary (1/0 instead of true/false) when needed.
3. Train a model based on the desired specifications (model type, smell, use or don't use grid search among others).

To Accomplish this I implemented what I see as a 2 level architecture with controllers that execute instructions using two classes called Model Generator and Model Evaluator. These two classes rely on the DecisionTree, RandomForest, NaiveBayes and SVC classes which inherit from a MLModel. Also, this two classes rely on the ModelFactory class which creates the appropriate model class instance to execute the instructions entered by the user. This inheritance gives these classes access to methods and other classes in charge of loading the data, cleaning the data , loading and persisting a model and finally evaluating the Accuracy and F1-score of a model. The architecture of the system can be seen in the attachments at the end of this document. Doing all of these things stopped me from duplicating the code needed to load the dataset and to load and persist a model.Once this design was in place adding the code for each model and testing them was faster.

Decision Tree Implementation.

To see my implementation of the Decision Tree models please look at the following files:

- **src/ml_models_engine/DataRepo.py**
- **src/ml_models/mlModels.py**
- **src/ml_models/DecisionTreeModel.py**

In the **DataRepo** class you will see that the **loadDataset** method does the following: first it loads the data into a dataframe using pandas, then it separates the data into dependent (Y) and independent(X) data, then the X data is filled with median values if there are empty or null values, and finally both X and Y data suffer certain transformations depending on the model that is using it. In the case of the Decision Tree I only convert the false and true values from the Y data into a multi label binary table. **Since this**

part of the process is similar or the same for all the models it was abstracted to a separate class to avoid code duplication.

Then if you go to mlModels you will find the methods that have code necessary to evaluate the accuracy and f1-score (precision and recall) of the model.

Finally in the DecisionTreeModel class you will find the code used to do the actions needed to train a decision tree. The class can use both the default parameters and grid search to search for the best model and best parameters needed to get the best model. I went with the GridSearch approach since this is the first time I trained a Decision Tree. The grid search relies on the following parameters to look for the best models:

- 10 folds to segregate the date
- 1, 5, 10, 20, 50 and 100 as the number of leaves that the tree should have.
- 1 - 21 levels of depth for the tree.
- The parameter used to evaluate the trees was accuracy

Random Forest Implementation.

To see my implementation of the Random Forest model please look at the following files:

- **src/ml_models_engine/DataRepo.py**
- **src/ml_models/mlModels.py**
- **src/ml_models/RandomForestModel.py**

In the **DataRepo** class you will see that the **loadDataset** method does the following: first it loads the data into a dataframe using pandas, then it separates the data into dependent (Y) and independent(X) data, then the X data is filled with median values if there are empty or null values, and finally both X and Y data suffer certain transformations depending on the model that is using it. In the case of the Decision Tree I only convert the false and true values from the Y data into a multi label binary table. **Since this part of the process is similar or the same for all the models it was abstracted to a separate class to avoid code duplication.**

Then if you go to mlModels you will find the methods that have code necessary to evaluate the accuracy and f1-score of the model.

Finally in the **RandomForestModel** class you will find the code used to do the actions needed to train a **Random Forest**. The class can use both the default parameters and grid search to search for the best model and best parameters needed to get the best model. I went with the GridSearch this model was also new to me. The grid search relies on the following parameters to look for the best models:

- 10 estimators with numbers between 10 and 70

- Auto and Square Root for the number of features the modeling can try in each individual tree.
- Max depth between 2 and 4
- Min sample split between 2 and 5
- Min sample leaves between 1 and 2

This GridSearch uses a RandomForestClassifier instance to find the best model and then this best model is stored in disk using joblib.

I didn't use a stratified K-fold cross validation like with the Decision Tree.

Naive Bayes Implementation.

To see my implementation of the Naive Bayes model please look at the following files:

- **src/ml_models_engine/DataRepo.py**
- **src/ml_models/mlModels.py**
- **src/ml_models/NaiveBayesModel.py**

In the **DataRepo** class you will see that the **loadDataset** method does the following: first it loads the data into a dataframe using pandas, then it separates the data into dependent (Y) and independent(X) data, then the X data is filled with median values if there are empty or null values, and finally both X and Y data suffer certain transformations depending on the model that is using it. In the case of the Decision Tree I only convert the false and true values from the Y data into a multi label binary table. **Since this part of the process is similar or the same for all the models it was abstracted to a separate class to avoid code duplication.**

Then if you go to mlModels you will find the methods that have code necessary to evaluate the accuracy and f1-score of the model.

Finally in the **NaiveBayesModel** class you will find the code used to do the actions needed to train a **Naive Bayes model**. Since the Naive Bayes algorithm is really simple and it is based on the Bayes theorem (therefore relying on the probability of something happening based on other known information and known previous outcomes) there are no hyper parameters to tune. However what I had to do was convert the Y data into 1 and 0 since the multi label format used before would not work in the math performed inside of this algorithm. This was done in the DataRepo class when loading the dataset . Also the NaiveBayesModel class has an attribute that indicates that Y data needs this transformation.

Support Vector Classifier Implementation.

To see my implementation of the Support Vector Classifier model please look at the following files:

- **src/ml_models_engine/DataRepo.py**
- **src/ml_models/mlModels.py**
- **src/ml_models/SVC.py**

In the **DataRepo** class you will see that the **loadDataset** method does the following: first it loads the data into a dataframe using pandas, then it separates the data into dependent (Y) and independent(X) data, then the X data is filled with median values if there are empty or null values, and finally both X and Y data suffer certain transformations depending on the model that is using it. In the case of the Decision Tree I only convert the false and true values from the Y data into a multi label binary table. **Since this part of the process is similar or the same for all the models it was abstracted to a separate class to avoid code duplication.**

Then if you go to mlModels you will find the methods that have code necessary to evaluate the accuracy and f1-score of the model.

Finally in the **SVCModel** class you will find the code used to do the actions needed to train a **Support Vector Classifier** with all the different kernels used to create the hyperplanes that separates the data.
I went with the default options, except for the polynomial kernel where I added degrees from 1 to 6 for the algorithm to use in the creation of the hyperplane.

However what I actually did in this algorithm was a feature scaling or normalization of the data for values between 0 and 1. This helped the algorithm perform drastically better. This was done based on multiple examples and videos[1] I saw that explain that for the SVC algorithm we have to normalize the data since usually it can have values with different magnitudes (and also different units) which is our case due to all the different metrics that we have.

[1] https://www.youtube.com/watch?v=mnKm3YP56PY

Part 2. Instructions to run the tool.

1. Install all the requirements on the Venv listed in requirements.txt.

2. Run the tool using the run.py script available in the root folder.

3. Commands available for the user:

Run python run.py --help to see the lists of commands. This will display the following list

```
jplozgom at jplozgom-3 in ~/Documents/jp/labWeb/cmu/semester3/dataScience/assignment1 (master)
$ python run.py --help
Usage: run.py [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  compare   Compare testing and training data metrics from a model
            generated...

  evaluate  Evaluate a model for multiple smells using models generated...
  models    Lists all the models available for the system and the user to...
  smells    Lists all the smells available for the system and the user to...
  train     Trains a model for a smell.
```

a. The command smells and models will list the models and smells available in the system.

```
jplozgom at jplozgom-3 in ~/Documents/jp/labWeb/cmu/semester3/dataScience/assignment1 (master•)
$ python run.py smells

1. DATA CLASS: A data class refers to a class that contains only fields and crude methods for accessing them (getters and setters).
2. FEATURE ENVY: A method accesses the data of another object more than its own data.
3. GOD CLASS: A class that knows too much or does too much. Also known as large class
4. LONG METHOD: A method contains too many lines of code. Usually more than 10
```

respective commands).

If you only want to train two
repeat the --smell option two

```
(assignment1_18668)
jplozgom at jplozgom-3 in ~/Documents/jp/labWeb/cmu/semester3/dataScience/assignment1 (master•)
$ python run.py models

1. DECISION TREE: Algorithm base on decision tree logic staring from the top and doing down to a leaf using conditions
2. RANDOM FOREST: Algorithm that implements multiple decision trees with different groups of data
3. NAIVE BAYES: Algorithm based on the Bayes theorem in conditional probability
4. SVC LINEAR: Support Vector Classifier using a linear kernel to create hyper planes
5. SVC POLYNOMIAL: Support Vector Classifier using polynomial kernels of n degrees to create hyper planes
6. SVC RBF: Support Vector Classifier using a Radial Basis function kernel to create hyper planes
7. SVC SIGMOID: Support Vector Classifier using a sigmoid (mathematical function having a characteristic 'S' -shaped cu
```

a. The commar

b. The commar
--model to indic
model in 1 pass

b. The command train will do the training of the models. For this enter the options --smell and --model to indicate the Classification model to be trained. If you want to train all the smells for a model in 1 pass you can enter **all** after --smells. Also if you want to see the outcome and some prints leave the --debug option in the command (these metrics are also available in their respective commands).

```
jplozgom at jplozgom-3 in ~/Documents/jp/labWeb/cmu/semester3/dataScience/assignment1 (master)
$ python run.py train --smell "all" --model "svc polynomial" --debug
```

If you only want to train two smells you can specify this by entering individually . To do this repeat the --smell option two times.

```
(assignment1_18668)
jplozgom at jplozgom-3 in ~/Documents/jp/labWeb/cmu/semester3/dataScience/assignment1 (master)
$ python run.py train --smell "Feature envy" --smell "god class" --model "svc polynomial" --debug
```

c. The evaluate and compare commands will display the accuracy and F1-score for the models and the smells. Evaluate does it for multiple smells and compare for training and test datasets.

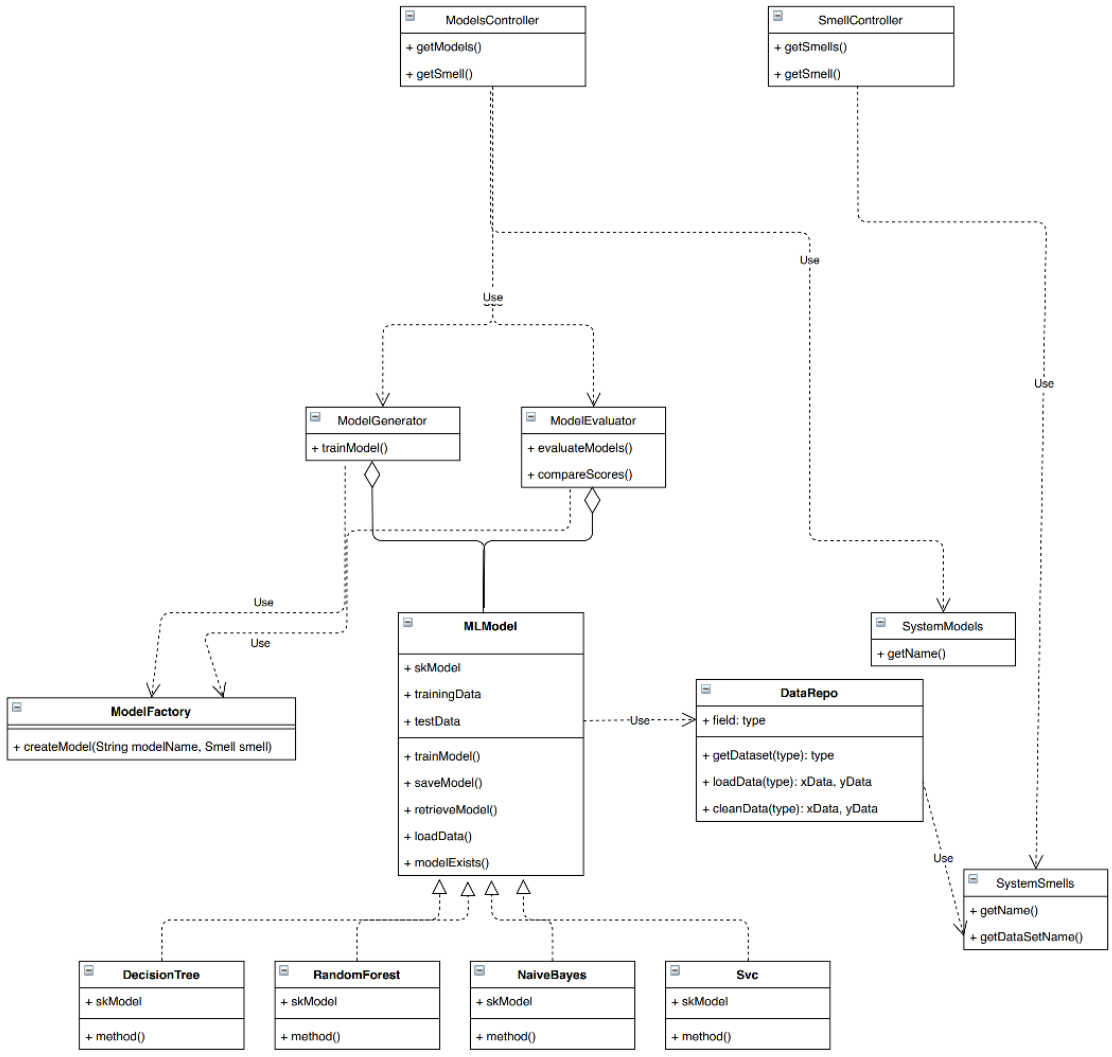
```
jplozgom at jplozgom-3 in ~/Documents/jp/labWeb/cmu/semester3/dataScience/assignment1 (master)
$ python run.py compare --smell "all" --model "decision tree"
```

```
jplozgom at jplozgom-3 in ~/Documents/jp/labWeb/cmu/semester3/dataScience/assignment1 (master)
$ python run.py evaluate --smell "all" --model "Decision tree"
```

Attachments

1. System Design.

This is the initial idea that I implemented. The end result changed in some methods but in general this diagram has the big picture of my system.



2. Model Performance metrics

```
$ python run.py evaluate --smell "all" --model "Decision tree"

Evaluation of a Decision Tree used to predict the smell following

Smell           Accuracy    F1-score
------------  ----------  ----------
data class      0.847339    0.887513
feature envy    0.841737    0.883933
god class       0.854342    0.894309
long method     0.833333    0.881356
```

```
$ python run.py evaluate --smell "all" --model "Random forest"

Evaluation of a Random Forest used to predict the smell following

Smell           Accuracy    F1-score
------------  ----------  ----------
data class      0.82493     0.875267
feature envy    0.833333    0.881356
god class       0.843137    0.887389
long method     0.833333    0.881356
```

```
$ python run.py evaluate --smell "all" --model "Naive Bayes"

Evaluation of a Naive Bayes used to predict the smell following

Smell           Accuracy    F1-score
------------  ----------  ----------
data class      0.567227    0.567227
feature envy    0.806723    0.806723
god class       0.838936    0.838936
long method     0.815126    0.815126
```

```
$ python run.py evaluate --smell "all" --model "SVC polynomial"

Evaluation of a SVC Polynomial used to predict the smell following

Smell           Accuracy    F1-score
------------  ----------  ----------
data class      0.836134    0.836134
feature envy    0.852941    0.852941
god class       0.858543    0.858543
long method     0.858543    0.858543
```

```
$ python run.py evaluate --smell "all" --model "SVC rbf"

Evaluation of a SVC RBF used to predict the smell following

Smell           Accuracy    F1-score
------------  ----------  ----------
data class      0.85014     0.85014
feature envy    0.837535    0.837535
god class       0.852941    0.852941
long method     0.838936    0.838936
```

Bibliography

https://medium.com/analytics-vidhya/accuracy-vs-f1-score-6258237beca2#:~:text=Accuracy%20is%20used%20when%20the,as%20in%20the%20above%20case.