Technical Report for Assignment - 4

Name: Siva Rama Rohan Sunkarapalli **Email:** ssunkarapalli@crimson.ua.edu

Statement: For this assignment's preparation, I have utilized ChatGPT, a language model created by OpenAI. Within this assignment, the ChatGPT was used for purposes such as brainstorming and grammatical corrections.

1. Introduction

In this assignment, we aim to build predictive models using both the TPOT (Tree based Pipeline Optimization Tool) AutoML tool, where limited intervention is necessary, and the traditional ML pipeline with careful model selection and hyperparameter tunings.

To accomplish the above comparison study, we selected the UCI Adult Income dataset (https://archive.ics.uci.edu/dataset/2/adult). The census dataset aims to predict whether incomes exceed 50,000.

The AutoML tools have gained popularity in the ML community to streamline model building, feature engineering, and hyperparameter tuning. This became possible because the ML evolute to follow common sequences with data preparation, model selection from trying several algorithms and selecting the one that yields the best performance metric.

In this comparative study, firstly, we load the dataset and then clean it by imputting missing values. Secondly, using the TPOT AutoML tool we build a train model. Thirdly, in comparison to the automatic ML, we create a trained model with hyperparameters tuning with traditional ML practices. At this step, we chose the Gradient Boosting Classifier model algorithm. Finally, we compare the performance, time consumption, and accuracy (chosen model evaluation metric).

2. Dataset Selection and Preprocessing

2.1. Dataset Description

The UCI Adult income dataset is a census data which aims to predict whether the income is more than 50,000 given the personal and professional characteristics. The dataset includes attributes including numerical and categorical attributes. Age, final weight, and education_num are numerical, and the remaining marital status, occupation, relationship, race, and sex are categorical. The authors have provided two datasets train and test with 32,561 and 16,281 samples respectively. The dataset is imbalanced with only 24% of the samples with positive labels - people with more than 50k annual income.

2.2 Preprocessing Techniques

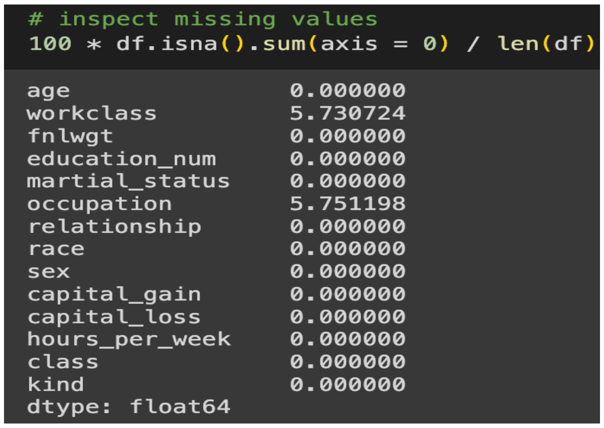
Since we have access to train and test datasets, we combined both and performed data preprocessing including cleaning missing values, standardization of numerical features, one-hot-encoding of categorical features.

- **Missing values**

The authors represented missing values with "?" and we iterated each feature and replaced it with null values for easier processing.

df = df.replace('?', np.nan)

The dataset is clean with only 5.7% missing values in the workclass and occupation columns.

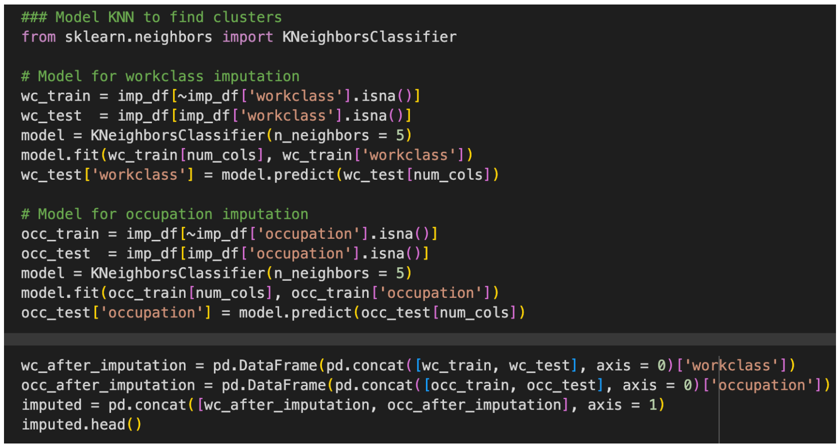


```
# inspect missing values
100 * df.isna().sum(axis = 0) / len(df)
```

```
age                 0.000000
workclass           5.730724
fnlwgt              0.000000
education_num       0.000000
martial_status      0.000000
occupation          5.751198
relationship        0.000000
race                0.000000
sex                 0.000000
capital_gain        0.000000
capital_loss        0.000000
hours_per_week      0.000000
class               0.000000
kind                0.000000
dtype: float64
```

Figure 1 – Inspecting Missing Values

The most common method is to impute the missing values with the most frequent labels, however we utilized KNN model to predict missing values given the numerical attributes.



```
### Model KNN to find clusters
from sklearn.neighbors import KNeighborsClassifier

# Model for workclass imputation
wc_train = imp_df[~imp_df['workclass'].isna()]
wc_test  = imp_df[imp_df['workclass'].isna()]
model = KNeighborsClassifier(n_neighbors = 5)
model.fit(wc_train[num_cols], wc_train['workclass'])
wc_test['workclass'] = model.predict(wc_test[num_cols])

# Model for occupation imputation
occ_train = imp_df[~imp_df['occupation'].isna()]
occ_test  = imp_df[imp_df['occupation'].isna()]
model = KNeighborsClassifier(n_neighbors = 5)
model.fit(occ_train[num_cols], occ_train['occupation'])
occ_test['occupation'] = model.predict(occ_test[num_cols])
```

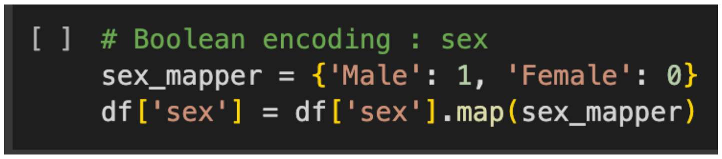
```
wc_after_imputation = pd.DataFrame(pd.concat([wc_train, wc_test], axis = 0)['workclass'])
occ_after_imputation = pd.DataFrame(pd.concat([occ_train, occ_test], axis = 0)['occupation'])
imputed = pd.concat([wc_after_imputation, occ_after_imputation], axis = 1)
imputed.head()
```

Figure 2 – Model KNN to Finding Clusters

In the above method, we defined an imp_df data frame with all the numerical features that are standardized using sklearn's StandardScaler method, and workclass and occupation features. The method aims to iteratively predict missing values using KNN algorithm by comparing 5 nearest neighbors.

- **Binarization**

In the original dataset, the authors labeled the class as <=50k and 50k, and we labeled them 0 and 1, respectively. Similarly, sex was mapped with 0 as female and 1 as male.

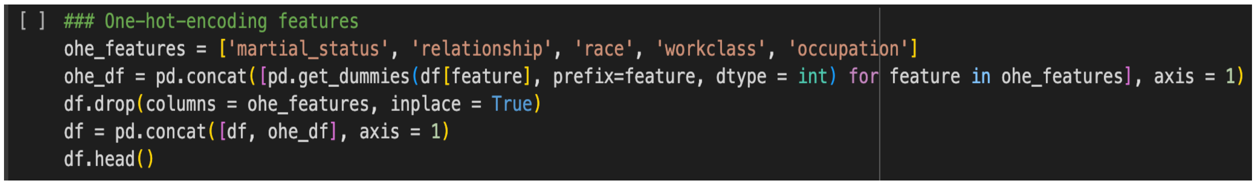


```
[ ]  # Boolean encoding : sex
     sex_mapper = {'Male': 1, 'Female': 0}
     df['sex'] = df['sex'].map(sex_mapper)
```

Figure 3 - Binarization

- **Data normalization and categorical encoding**

The numerical features are standardized with sklearn's StandardScaler method where they are centered around 0 and dispersed with standard deviation. The remaining categorical features including marital status, relationship, race, workclass, and occupation are encoded with one-hot-encoding method.



```
[ ]  ### One-hot-encoding features
     ohe_features = ['martial_status', 'relationship', 'race', 'workclass', 'occupation']
     ohe_df = pd.concat([pd.get_dummies(df[feature], prefix=feature, dtype = int) for feature in ohe_features], axis = 1)
     df.drop(columns = ohe_features, inplace = True)
     df = pd.concat([df, ohe_df], axis = 1)
     df.head()
```

Figure 4 – Encoding Features

Total number of features after cleaning, normalization, and encoding is 47. We will use them to make predictions for the class - whether the income exceeds 50,000.

3. AutoML Implementation

3.1. AutoML Tool Selection

The library we selected for the AutoML is TPOT. Its workflow is shown below. Briefly, the framework takes cleaned and preprocessed input data (including both features and target values) and then performs feature selection, feature engineering in parallel before passing into various models for model selection while optimizing the past scoring metric. The framework splits the data into multiple folds for better generalization - reducing overfitting.

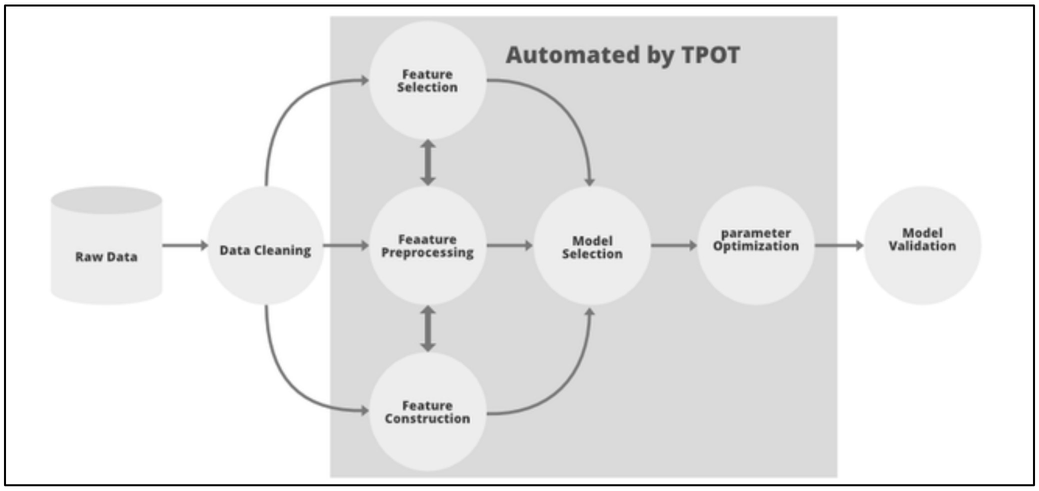
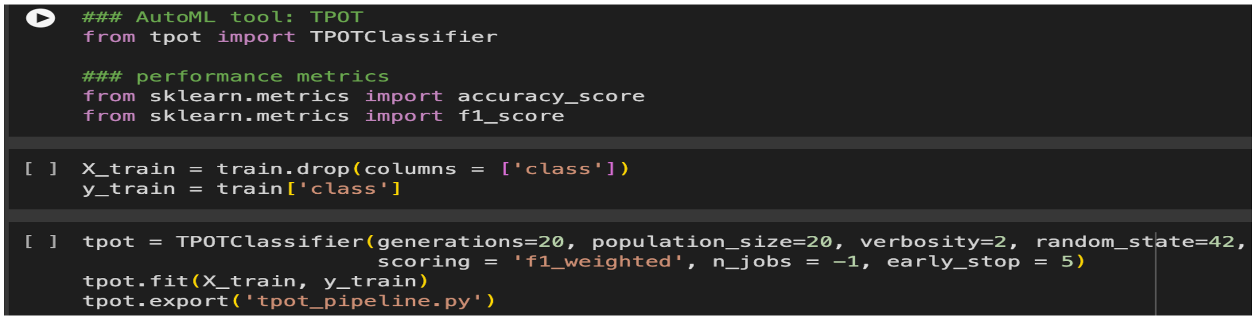


Figure 5 – Workflow of AutoML

Workflow of AutoML library of TPOT (https://www.geeksforgeeks.org/tpot-automl/)

3.2. Model Training and Selection

Implementation of the framework for the classification problem is shown below.



```
### AutoML tool: TPOT
from tpot import TPOTClassifier

### performance metrics
from sklearn.metrics import accuracy_score
from sklearn.metrics import f1_score
```

```
X_train = train.drop(columns = ['class'])
y_train = train['class']
```

```
tpot = TPOTClassifier(generations=20, population_size=20, verbosity=2, random_state=42,
                      scoring = 'f1_weighted', n_jobs = -1, early_stop = 5)
tpot.fit(X_train, y_train)
tpot.export('tpot_pipeline.py')
```

Figure 6 - Model Training and Selection

After importing the TPOTClassifier from the tpot package, we instantiate to run 20 iterations where several model architectures are trained and evaluated for the past scoring metric (f1_weighted). The automation process was time consuming and with early stop (5), we made it terminate the search if the score did not improve for 5-successive generations. In the end, the model architecture is saved in the tpot_pipeline.py file. The evaluation metric (f1_weighted) selected imbalance in the target class with only 24% sample with label 1.

Best pipeline:

```
        GradientBoostingClassifier(input_matrix,
        learning_rate=0.1,
        max_depth=8,
        max_features=0.35000000000000003,
        min_samples_leaf=3,
        min_samples_split=17,
        n_estimators=100,
        subsample=0.8)
```

The best model for the AutoML framework selected was Gradient Boosting classifier with the hyperparameters shown above.

3.3. Performance Evaluation

The best internal CV score after running 20 generations is 0.8686 weighted f1-score with the corresponding accuracy of 90.08 percent. The corresponding accuracy score on the test dataset is 87.06 percent.

4. Comparison with Traditional Methods

4.1. Traditional Machine Learning Pipeline

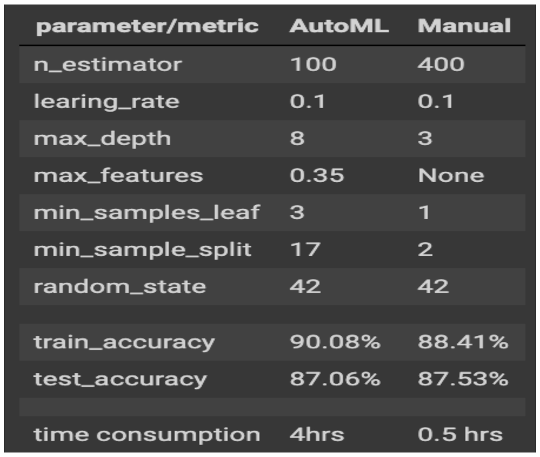
The same Gradient Boosting Classifier was selected and implemented manual tuning of hyperparameters iteratively. The following table shows a list of hyperparameters and the corresponding search space along with best values.

Hyperparameter	**Search Space**	**Best**
n_estimators	200, 300, 400, 500, 1000	400
learning_rate	0.001, 0.01, 0.1, 0.2, 0.5	0.1
max_depth	2, 3, 5, 7	3
max_features	None, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, ‘sqrt,’ ‘log2’	None
min_samples_leaf	1, 3, 5, 7, 9, 11, 13	1
min_samples_split	2, 4, 8, 16, 32	2
random_state	42, 512, 2048, 4096	42

Before hyper tuning, baseline performance of the model was established with default parameter values and achieved train accuracy of 86.88% and test accuracy of 86.94%. The test performance is not significantly different from that of AutoML (87.06%). After manual parameter tuning (iteratively), the test accuracy increased to 87.52 percent.

4.2. Comparative Analysis

The following table compares the performance of manual hyper tuning with AutoML. We observe marginal improvement in test accuracy and lower difference in train and test performance. The most significant difference between the two approaches is the process time. It took about 4 hours for the AutoML to complete all the generations, whereas we were able to fine tune hyperparameters in 30 minutes. It is important to note that we started with the Gradient Boosting model in the manual ML, had we tried several algorithms, the process time is expected to increase considerably.



parameter/metric	AutoML	Manual
n_estimator	100	400
learing_rate	0.1	0.1
max_depth	8	3
max_features	0.35	None
min_samples_leaf	3	1
min_sample_split	17	2
random_state	42	42
train_accuracy	90.08%	88.41%
test_accuracy	87.06%	87.53%
time consumption	4hrs	0.5 hrs

Figure 7- Comparative Analysis

5. Analysis

5.1 Benefits and Limitations of AutoML

Benefits of AutoML –

- It tries various ML models and selects the model algorithm with best score.
- It performs feature selection, feature engineering, and hyperparameter tuning as a part of the process.

Limitations of AutoML -

- It offers limited control on the preselecting of the limited number of models and range of hyperparameters in the tuning step.
- The framework iterates a limited number of times (given in the generations argument) and does not guarantee to result in the best score.
- The tool consumes a large amount of time as it performs various iterations, however, early stopping arguments help but we cannot guarantee that the framework reached the best performing model / hyperparameters.

The TPOT AutoML tool tries various ML models including random forest, gradient boosting, and xgboost, among the few. I was expecting the tool to select XgBoost for its expected better performance, but it selected GradientBoostingClassifier.

5.2 Reflection on Model Choices and Hyperparameters

In the comparative study, we demonstrated the implementation of the AutoML framework from TPOT on the cleaned and preprocessed UCI Adult Income dataset. The framework selected Gradient Boosting algorithm and performed hyperparameter tuning. Later, we took the same model algorithm and tuning parameters manually. The difference in performance was marginal, however the significant difference was in process time. Overall, the manual process yielded a better accuracy score of 87.53 percent.

6. Lessons Learned

Comparing the pipeline of AutoML with the TPOT framework and traditional ML pipeline we learned that they both have pros and cons. For the selected dataset and problem, we defined for classification, we can see that AutoML plays a vital role in model algorithm selection, while the traditional way of hyperparameter tuning results in better performance. It is recommended to try AutoML for model selection and perform hyperparameter tuning with either iteratively or even better with random grid search.

7. References

- Dataset: https://archive.ics.uci.edu/dataset/2/adult
  AutoML TPOT: http://epistasislab.github.io/tpot/api/