Title

Report for the CEGE0004: Assignment

**─**

Group Name

Your Name 1

Your Portico Number 1

Your Email 1

Your Name 2

Your Portico Number 2

Your Email 2

Your Name 3

Your Portico Number 3

Your Email 3

Your GitHub Repo Link

Use the text highlighted in orange as a guideline and delete it before submitting the report.

General recommendations:

* Delete the sections that you have not completed.
* You can copy and paste functions/methods/classes/bash-commands here and describe how they work. Please don’t copy and paste screenshots (penalties will occur if you do that).

# Introduction

Describe the problem and look at the big picture.

This section should answer the following questions:

* Why would anybody care about this task? Motivate your work.
* Which learning tasks are you trying to solve?
* How should performance be measured?
* How would you solve this problem manually?

# Learning Tasks

Formalize mathematically the tasks you solved. For each task, you should clearly describe what kind of features your model will deal with and what is the expected output.

# Material

Describe the data.

Do some data exploration.

Study each attribute and its characteristics:

* Name
* Type (categorical, int/float, bounded/unbounded, text, structured, etc.)
* % of missing values
* Noisiness and type of noise (stochastic, outliers, rounding errors, etc.)
* Type of distribution (Gaussian, uniform, logarithmic, etc.)

Identify the target attribute(s).

Visualize the data.

Study the correlation between attributes.

Identify any common (among the tasks) transformations you may want to apply.

Describe how you sampled the test set.

# Technology

Describe any technology you used in order to solve this assignment.

# Learning Algorithms

## Decision Trees

Give a general introduction to this family of algorithms and the specific one you picked. Describe which task you solved using this algorithm. Present any specific preprocessing step you needed to perform. Describe how you implemented this algorithm, how you tuned its hyperparameters, and validate its performance.

## Instance-Based Learning

Instance-Based Learning is a family of supervised learning methods that simply store the training examples. Instance-based methods are sometimes referred to as "lazy" learning methods because they delay processing until a new instance must be classified. A key advantage of this kind of delayed, or lazy, learning is that instead of estimating the target function once for the entire instance space, these methods can estimate it locally and differently for each new instance to be classified [1].

There are few instance-based learning algorithms, such as:

1. K-nearest Neighbors (KNN)
2. Collaborative filtering
3. Support Vector Machine (SVM)

The dataset that would be use for the assignment is Online Shoppers Purchasing Intention [2] (detail discussion on the dataset and how to pre-process the data are discussed on "Material" section). The dataset is pre-processed to just include features that are considered have impacts on the outcomes. The dataset's features are either numerical/continuous (4 features) or categorical (5 features) with total data of 12,330 records so the dimensionality is considered low. The dataset only has 2 classes (not multi-class). The instance-based learning algorithms that would be explored for the dataset is KNN and SVM.

KNN is a very simple, easy to understand, versatile and one of the topmost machine learning algorithms. KNN calculates the approximate distances between the vectors and then assign the points which are not yet labelled to the class of its k-nearest neighbours. In KNN, K is the number of nearest neighbours. The number of neighbours is the core deciding factor. K is generally an odd number if the number of classes is 2. KNN performs better with a lower number of features than a large number of features [3].

KNN has the following basic steps:

1. Calculate distance
2. Find closest neighbors
3. Vote for labels

SVM offers very high accuracy compared to other classifiers such as logistic regression and decision trees. SVM works with the classifier separates data points using a hyperplane with the largest amount of margin. SVM finds an optimal hyperplane which helps in classifying new data points. The idea behind SVMs is to maximize the margin that better classifies the data. The hyperparameter 𝐶 set when training an SVM classifier is a tradeoff between encouraging larger margins by violating them or being stricter with the violations by making the margins smaller [4]. SVM searches for the maximum marginal hyperplane in the following steps [3]:

1. Generate hyperplanes which segregates the classes in the best way.
2. Select the right hyperplane with the maximum segregation from the either nearest data points.

### K-nearest Neighbors

The performance of the K-NN algorithm is influenced by three main factors:

1. The distance function or distance metric used to determine the nearest neighbors.
2. The decision rule used to derive a classification from the K-nearest neighbors.
3. The number of neighbors used to classify the new example.

To implement the classifier, the KNN algorithm from the scikit learn module was used (sklearn.neighborsKNeighborsClassifier). The parameter we need to tuning is weights, n\_neighbors, and metric. We use GridSearchCV to select the best parameters from the listed hyperparameters. GridSearchCV is a library function that is a member of sklearn’s model\_selection package. It helps to loop through predefined hyperparameters and fit the estimator (model) on your training set. The grid search picks out a grid of hyperparameter values, evaluates every one of them, and returns the winner. We can also determine the number of times for cross-validation for each set of hyperparameters.

We split the dataset into a set for training (75%) and testing (25%) (stratified to ensure they both have the same class ratio). Then, a cross-validated grid-search method (GridSearchCV) over a manually set parameter grid was conducted to tune the hyperparameters and evaluate the models. The cross validation split the data into *𝑘* (typically 5 or 10 is generally optimal, but due to the smaller size of the dataset, 5 was selected to maintain representativeness of the samples) folds, where the data was fitted using *𝑘*−1 folds and validated with the remaining fold for each parameter set. Therefore, we do not split the dataset into testing, validation and test. The GridSearchCV using cv=5 would do the work.

The hyperparameter setting returning the highest average f1 score was then selected as the model for use on the test set. The performance metric used is 'f1', which is a weighted average of the precision and recall of the model [5]. The reason for choosing f1 score over the more commonly used 'Accuracy' is because of the class imbalance (high number of majority class 'False' examples), as the f1 score does not make use of the 'True Negatives' as with the case of 'Accuracy', but rather focuses on the ability of the model to predict the minority class correctly. The following codes show the function for searching classification with KNN using GridSearchCV and the output.

def src\_classifier\_knn(x\_train, y\_train):  
 param\_grid = [{  
 'weights': ["uniform", "distance"],  
 # 'n\_neighbors': range(k\_min,k\_max),  
 'n\_neighbors': [1, 3, 5],  
 'metric': ['euclidean', 'manhattan', 'cosine', 'minkowski', 'hamming']}]  
  
 classifier = KNeighborsClassifier()  
 grid\_search = GridSearchCV(classifier, param\_grid, cv=5, verbose=0, scoring='f1')  
 grid\_search.fit(x\_train, y\_train)  
  
  
 classifier = grid\_search.best\_estimator\_  
 print(classifier)  
 print(grid\_search.best\_params\_)  
  
 classifier.fit(x\_train, y\_train)

return classifier

KNeighborsClassifier(metric='cosine', n\_neighbors=1)

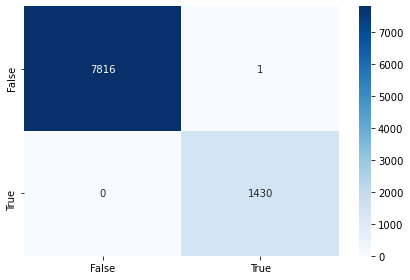
{'metric': 'cosine', 'n\_neighbors': 1, 'weights': 'uniform'}

Accuracy Report for Training

Train score 0.9998918568184276

F1 Score: 1.00

Train Confusion matrix



False True

precision 0.999872 1.000000

recall 1.000000 0.999301

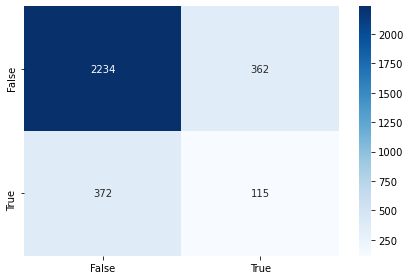
print("Accuracy Report for Testing")  
y\_pred\_test = prediction(classifier, x\_test)  
accuracy\_cm\_report(y\_test, y\_pred\_test, class\_names=class\_names)

Accuracy Report for Testing

Train score 0.7619202075900098

F1 Score: 0.24

Train Confusion matrix



False True

precision 0.860555 0.23614

recall 0.857252 0.24109

We can see that the GridSearchCV result for the best parameters are {'metric': 'cosine', 'n\_neighbors': 1, 'weights': 'uniform'} based on highest average f1 score for the training dataset. We can see the f1 score for training dataset is 1.00 which is considered very high. However, using the same parameters, the f1 score for test dataset is 0.24 which is considered low. This is might be caused by the selected n\_neighbours which is 1 make the classification overfit. There are some numbers that are tried as n\_neighbour but after being compared with n\_neighbour=1, the best parameter for n\_neighbour is always one. It might also be caused by the dataset itself that is highly unbalanced, where the 'False' class outnumbers the 'True' class by a ratio of about 5:1. There is also possibility that the model does not perform very well because the dataset is not normalized. We will look how does SVM perform on the dataset.

### Support Vector Machine

Same as KNN or other machine learning algorithms, SVM also need to tune the parameters’ values to effectively improves model performance. There are some important parameters that having higher impact on model performance which are:

1. kernel, specifies the kernel type to be used in the algorithm. It must be one of ‘linear’, ‘poly’, ‘rbf’, ‘sigmoid’, ‘precomputed’ or a callable.
2. gamma, kernel coefficient for ‘rbf’, ‘poly’ and ‘sigmoid’.
3. C, is regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive.

To implement the classifier, the SVM algorithm from the scikit learn module was used (sklearn.svm.SVC). As discussed earlier, the parameter we need to tune are kernel, gamma, and C. We use GridSearchCV to select the best parameters from the listed hyperparameters. We use GridSearchCV to select the best parameters from the listed hyperparameters that has been discussed earlier.

We tried to search the classifier with the same dataset and scoring f1 as discussed earlier. However, using the dataset, GridSearchCV takes so much time and do not show result. Therefore, we perform another pre-processing of the dataset.

The first pre-processed dataset "data\_frame\_os" represents the dataset in which the categorical variables have been factorized whilst the numerical variables are left untouched. We performed another pre-process of the dataset "data\_frame\_os" to become "data\_frame\_os\_cat" that represents the transformed dataset in which all variables have been set to categorical. Then, We performed GridSearchCV for SVM with new dataset "data\_frame\_os\_cat". The following codes show the function for searching classification with SVM using GridSearchCV following with the output.

def src\_classifier\_svm(x\_train, y\_train):

parameters = {'kernel': ('linear', 'rbf', 'sigmoid'), 'gamma': ('scale',

'auto'), 'C': (7, 8, 13)}

svc = SVC()  
 grid\_search = GridSearchCV(svc, parameters, cv=5, verbose=0)  
 grid\_search.fit(x\_train, y\_train)  
  
 classifier = grid\_search.best\_estimator\_  
 print(classifier)  
 print(grid\_search.best\_params\_)  
  
 classifier.fit(x\_train, y\_train)  
 return classifier

SVC(C=8, kernel='sigmoid')

{'C': 8, 'gamma': 'scale', 'kernel': 'sigmoid'}

Accuracy Report for Training

Train score 0.7572185573699578

F1 Score: 0.21

Train Confusion matrix

False True

precision 0.856157 0.214586

recall 0.856704 0.213836

print("Accuracy Report for Testing")  
y\_pred\_test = prediction(classifier, x\_test)  
accuracy\_cm\_report(y\_test, y\_pred\_test, class\_names=class\_names)

Accuracy Report for Testing

Train score 0.7421342847875446

F1 Score: 0.19

Train Confusion matrix

False True

precision 0.851378 0.185771

recall 0.841903 0.197065

We can see that the GridSearchCV result for the best parameters are {'C': 8, 'gamma': 'scale', 'kernel': 'sigmoid'} based on highest average f1 score for the training dataset. We can see the ‘f1’ scores for training and test dataset are both less than or equal to 0.21 which is considered low. It might be caused by the dataset itself that is highly unbalanced, where the 'False' class outnumbers the 'True' class by a ratio of about 5:1. There is also possibility that the model does not perform very well because the dataset is not normalized.

In this report, we tried to cover the imbalanced data issue by performing resampling technique. Resampling techniques may be helpful in helping to address the issue of class imbalance [6]. There are three methods that are tested in this report which are up-sampling and down-sampling.

Up-sampling adds more copies of the minority class in the training set to make it even with the majority class, while conversely down-sampling removes observations of the majority class. We cannot use the whole dataset when performing resample. We need to divide the dataset into training set and test set before performing resample. The resampling module from Scikit-Learn has been used to achieve this.

### Up-sampling

The followings are codes to up sampling the dataset (true value) and the outputs.

def upsample\_minority(x, y):  
 # concatenate our training data back together  
 X = pd.concat([x, y], axis=1)  
  
 # separate minority and majority classes  
 not\_true = X[X.Revenue == 0]  
 true = X[X.Revenue == 1]  
  
 # upsample minority  
 true\_upsampled = resample(true,  
 replace=True, # sample with replacement  
 n\_samples=len(not\_true), # match number in majority class  
 random\_state=43) # reproducible results  
  
 # combine majority and upsampled minority  
 upsampled = pd.concat([not\_true, true\_upsampled])  
  
 # check new class counts  
 upsampled.Revenue.value\_counts()  
  
 y = upsampled.Revenue  
 x = upsampled.drop('Revenue', axis=1)  
  
 return x, y

KNeighborsClassifier(metric='hamming', n\_neighbors=3, weights='distance')

{'metric': 'hamming', 'n\_neighbors': 3, 'weights': 'distance'}

Accuracy Report for Training with KNN

Train score 0.9318065506653019

F1 Score: 0.93

False True

precision 0.952656 0.912794

recall 0.908777 0.954836

Accuracy Report for Testing KNN

Train score 0.70969834576711

F1 Score: 0.31

False True

precision 0.877371 0.243873

recall 0.763239 0.417191

​

Accuracy Report for Testing KNN

Train score 0.70969834576711

F1 Score: 0.31

Train Confusion matrix

False True

precision 0.877371 0.243873

recall 0.763239 0.417191

def src\_classifier\_svm(x\_train, y\_train):  
 parameters = {'kernel': ('linear', 'rbf'), 'C': (15, 17, 19)}  
 svc = SVC()  
 grid\_search = GridSearchCV(svc, parameters, cv=5, verbose=0)  
 grid\_search.fit(x\_train, y\_train)  
  
 classifier = grid\_search.best\_estimator\_  
 print(classifier)  
 print(grid\_search.best\_params\_)  
  
 # classifier = SVC(kernel='sigmoid', gamma='scale', C=8)  
 # print(classifier)  
  
 classifier.fit(x\_train, y\_train)  
 return classifier

SVC(C=19)

{'C': 19, 'kernel': 'rbf'}

Accuracy Report for Training with SVM

Train score 0.7250511770726714

F1 Score: 0.75

Train Confusion matrix

False True

precision 0.787418 0.684924

recall 0.616556 0.833547

Accuracy Report for Testing SVM

Train score 0.6143366850470321

F1 Score: 0.37

False True

precision 0.927064 0.250000

recall 0.590177 0.746331

We can see that the GridSearchCV result changes for the up sampled dataset both for KNN and SVM. We can also see the increase of f1 scores for testing with KNN and SVM. Overall, after up sampling, SVM performs better than KNN for test data. There are many numbers that have been tried either kernel, gamma and C parameter. The SVM and the "f1" score increase as "C" increase. However, the searching time is also increase. Therefore, we selected parameters with acceptable processing time.

Down-sampling

The followings are codes to up sampling the dataset (non-true value) and the outputs.

def downsample\_majority(x, y):  
 # concatenate our training data back together  
 X = pd.concat([x, y], axis=1)  
  
 # separate minority and majority classes  
 not\_true = X[X.Revenue == 0]  
 true = X[X.Revenue == 1]  
  
 # downsample minority  
 not\_true\_downsampled = resample(not\_true,  
 replace=False, # sample without replacement  
 n\_samples=len(true), # match minority n  
 random\_state=42) # reproducible results  
  
 # combine majority and upsampled minority  
 downsampled = pd.concat([not\_true\_downsampled, true])  
  
 # check new class counts  
 downsampled.Revenue.value\_counts()  
  
 y = downsampled.Revenue  
 x = downsampled.drop('Revenue', axis=1)  
  
 return x, y

KNeighborsClassifier(metric='euclidean')

{'metric': 'euclidean', 'n\_neighbors': 5, 'weights': 'uniform'}

Accuracy Report for Training with KNN

Train score 0.7624039133473096

F1 Score: 0.77

False True

precision 0.775092 0.750835

recall 0.739343 0.785465

Accuracy Report for Testing KNN

Train score 0.7619202075900098

F1 Score: 0.24

Train Confusion matrix

False True

precision 0.860555 0.23614

recall 0.857252 0.24109

SVC(C=19)

{'C': 19, 'kernel': 'rbf'}

Accuracy Report for Training with SVM

Train score 0.7204751921733054

F1 Score: 0.75

Train Confusion matrix

False True

precision 0.781948 0.68101

recall 0.611461 0.82949

Accuracy Report for Testing KNN

Train score 0.5806033084657801

F1 Score: 0.37

Train Confusion matrix

False True

precision 0.933333 0.239796

recall 0.542594 0.788260

We can see that the GridSearchCV result changes for the down-sampled dataset both for KNN and SVM. We can also see the f1 scores for testing with KNN and SVM with down-sample is around the same with up-sample.

## Bayesian Learning

Give a general introduction to this family of algorithms and the specific one you picked. Describe which task you solved using this algorithm. Present any specific preprocessing step you needed to perform. Describe how you implemented this algorithm, how you tuned its hyperparameters, and validate its performance.

## Neural Networks

Give a general introduction to this family of algorithms and the specific one you picked. Describe which task you solved using this algorithm. Present any specific preprocessing step you needed to perform. Describe how you implemented this algorithm, how you tuned its hyperparameters, and validate its performance.

## Model Ensemble

Give a general introduction to the ensemble method you chose and make an ensemble of the models above for a shared task. Describe which task you solved using this algorithm. Present any specific preprocessing step you needed to perform. Describe how you implemented this algorithm, how you tuned its hyperparameters, and validate its performance.

# Conclusion

Present a comparison of the models studied above on the test results and make some hypotheses about why you observed these behaviors.

# Git Log

Copy and paste here the output of git log.