**FML (Fundamentals of Machine Learning) Lab**

**Nikhil S G (241058024)**

**Lab 1: Introduction to Python and Basic Syntax. Jul 30, 2024**

**Lab 2: User Input, Convert User Input, Comparison Operators, Exception Handling, Try-Except Structure, Functions, Loops and Iterations. Aug 6, 2024**

**Lab 3: String Manipulation. Aug 13, 2024**

**Lab 4: Lists, Tuples, and Dictionaries, Time, lambda function. Aug 20, 2024**

**Lab 5: Data Pre-processing in Python. Sep 3, 2024**

In this lab, we focused on data pre-processing techniques using Python, which are crucial steps before applying machine learning algorithms.

Step 1: Data Import and Overview

import numpy as np

We first import the necessary libraries and load a CSV file (Data.csv) from Google Drive using pandas. The head() function is used to preview the first few rows of the dataset.

import pandas as pd

import matplotlib.pyplot as plt

import sklearn

from sklearn.impute import SimpleImputer

from google.colab import drive

# Mount Google Drive to access the dataset

drive.mount('/content/drive')

# Load the dataset

data = pd.read\_csv('/content/drive/MyDrive/BDA\_1\_Sem/FML/FML\_Lab/Datasets/Data.csv')

dataset = data

dataset.head()

Step 2: Extract Features and Labels

Here, we split the data into **features** (X) and **labels** (Y). Features (X) are all the columns except the last one, while labels (Y) are the last column (Purchased). using pandas. The head() function is used to preview the first few rows of the dataset.

# Extract features (X) and labels (Y)

X = dataset.iloc[:, :-1].values # Features (all columns except the last one)

Y = dataset.iloc[:, -1].values # Label (last column)

Step 3: Handling Missing Data

The SimpleImputer from sklearn is used to handle **missing data**. Here, we use the **mean imputation strategy** to replace missing values (NaN) in the Age and Salary columns. The missing values are filled with the average value of the respective columns.preview the first few rows of the dataset.

# Impute missing data using mean strategy

from sklearn.impute import SimpleImputer

imputer = SimpleImputer(missing\_values=np.nan, strategy='mean')

imputer.fit(X[:, 1:3]) # Apply to 'Age' and 'Salary' columns

X[:, 1:3] = imputer.transform(X[:, 1:3])

Step 4: Encoding Categorical Data

**Label Encoding:** The **LabelEncoder** is used to convert the categorical feature (Country) into numeric labels (e.g., 'France' → 0, 'Spain' → 2, 'Germany' → 1).

**One-Hot Encoding**: After label encoding, we use **One-Hot Encoding** via ColumnTransformer to create dummy variables for the Country column (e.g., create separate columns for 'France', 'Spain', and 'Germany'). This ensures the model can understand categorical variables in numeric form, preventing the algorithm from mistakenly assuming any ordinal relationship between categories.

from sklearn.preprocessing import LabelEncoder, OneHotEncoder

from sklearn.compose import ColumnTransformer

# Label encoding for 'Country' (converting categorical values to numeric)

labelencoder\_X = LabelEncoder()

X[:, 0] = labelencoder\_X.fit\_transform(X[:, 0])

# OneHotEncoding for 'Country' (creating dummy variables)

ct = ColumnTransformer([("Country", OneHotEncoder(), [0])], remainder='passthrough')

X = ct.fit\_transform(X)

Step 5: Encoding the Dependent Variable

The dependent variable (Purchased) is categorical, with values "Yes" and "No". We apply **Label Encoding** to convert these categorical labels into numeric values (e.g., 'Yes' → 1, 'No' → 0), so they can be used in machine learning models.

# Label encoding for the dependent variable 'Purchased' (Yes/No)

labelencoder\_y = LabelEncoder()

Y = labelencoder\_y.fit\_transform(Y)

We split the dataset into a training set (80%) and a test set (20%) using the train\_test\_split function. This ensures that we have data to train the model on (X\_train, Y\_train) and data to evaluate its performance (X\_test, Y\_test).

Step 6: Splitting the Dataset into Training and Test Sets

from sklearn.model\_selection import train\_test\_split

# Split the dataset into training and test sets (80% train, 20% test)

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, Y, test\_size=0.2, random\_state=0)

Step 7: Feature Scaling

Since the features have different units (e.g., age in years, salary in dollars), we standardize them using StandardScaler. This ensures all features have a mean of 0 and a standard deviation of 1. This step is important for algorithms like logistic regression and neural networks, which are sensitive to the scale of the data.

from sklearn.preprocessing import StandardScaler

# Feature scaling (Standardization)

sc\_X = StandardScaler()

X\_train = sc\_X.fit\_transform(X\_train)

X\_test = sc\_X.transform(X\_test)

Lab Summary:

These pre-processing steps ensure that the dataset is clean, the features are appropriately encoded, and the model is ready for training. This is a crucial step in building machine learning models to ensure accuracy and performance.

**Lab 6: Classification and Regression. Sep 24, 2024**

**Part A**, we focused on pre-processing the **Social Network Ads dataset** and building a classification model to predict whether a user is likely to purchase a product based on their **age** and **estimated salary**.

Step 1: Data Import

We loaded the **Social\_Network\_Ads** dataset that contains years of experience and corresponding salaries.

# Load the dataset from Google Drive

data = pd.read\_csv('/content/drive/MyDrive/BDA\_1\_Sem/FML/FML\_Lab/Datasets/Social\_Network\_Ads.csv')

dataset = data

dataset.head()

Step 2: Extract Features and Labels (Same code from data pre-processing)

Step 3: Splitting the Data into Training and Testing Sets (Same code from data pre-processing)

Step 4: Feature Scaling (Same code from data pre-processing)

Step 5: Building a Decision Tree Classifier

We used a Decision Tree Classifier to build the model. The criterion='entropy' specifies the use of information gain as the measure for splitting nodes.

The classifier is trained using the X\_train and Y\_train data, where it learns to predict whether a user will purchase the product based on the given features (age and salary).

from sklearn.tree import DecisionTreeClassifier

classifier = DecisionTreeClassifier(criterion='entropy', random\_state=0) classifier.fit(X\_train, Y\_train)

The trained decision tree model is used to make predictions on the test set (X\_test), and the predicted labels are stored in Y\_pred. These predictions represent the model's output for whether each user in the test set is expected to purchase the product.

Step 6: Making Predictions

Y\_pred = classifier.predict(X\_test)

62 correct predictions for no purchase (True Negatives)

29 correct predictions for purchase (True Positives)

6 false positives (users predicted to purchase but did not)

3 false negatives (users predicted to not purchase but did)

[[62, 6],

[ 3, 29]]

Step 7: Evaluating Model Performance

from sklearn.metrics import confusion\_matrix

cm = confusion\_matrix(Y\_test, Y\_pred)

We calculated the accuracy of the model using accuracy\_score, which measures the proportion of correctly classified instances.

The accuracy of the model is 91%, meaning 91% of the predictions were correct, indicating a good model performance.

Step 8: Accuracy Calculation

from sklearn.metrics import accuracy\_score

prediction = accuracy\_score(Y\_test, Y\_pred)

print("Accuracy : ", prediction\*100, "%")

In **Part B**, the task was focused on **simple linear regression**, where we aimed to predict **Salary** based on **Years of Experience**.

Step 1: Data Import

We loaded the **Salary Data** dataset that contains years of experience and corresponding salaries.

data = pd.read\_csv('/content/drive/MyDrive/BDA\_1\_Sem/FML/FML\_Lab/Datasets/Salary\_Data.csv')

dataset = data

dataset.head()

Step 2: Extract Features and Labels (Same code from data pre-processing)

Step 3: Splitting the Data into Training and Testing Sets (Same code from data pre-processing)

Step 4: Model Training (Linear Regression)

We trained a **Simple Linear Regression model** to predict Salary based on Years of Experience.

from sklearn.linear\_model import LinearRegression

regressor = LinearRegression()

regressor.fit(X\_train, Y\_train)

We used the trained model to predict the **Salary** on the test set.

Step 5: Prediction

Y\_pred = regressor.predict(X\_test)

Step 6: Visualization

We visualized the results by plotting both the training and testing data points along with the regression line. The training data is shown in red, testing data in green, and the regression line is in blue.

plt.scatter(X\_train, Y\_train, color='red') # Training data points

plt.scatter(X\_test, Y\_test, color='green') # Test data points

plt.plot(X\_train, regressor.predict(X\_train), color='blue') # Regression line

plt.title('Salary vs Experience (Training Set)')

plt.xlabel('Years of Experience')

plt.ylabel('Salary') plt.show()

Lab Summary:

Supervised Learning:

Classification (Part A) and Regression (Part B) are both types of supervised learning where we learn from labeled data.

In Part A, we performed classification of data points using a Decision Tree Classifier.

In Part B, we performed regression to predict a continuous target variable (Salary) based on a feature (Years of Experience) using Linear Regression.

**Lab 7: Logistic Regression. Oct 1, 2024**

we applied **Logistic Regression** for classification tasks with the **Social Network Ads** dataset to predict whether a user would purchase a product based on their **Age** and **Estimated Salary**.

Step 1: Data Import (same as the Lab 6 Part A **Social Network Ads dataset)**

Step 2: Splitting the Data into Training and Testing Sets (Same code from data pre-processing)

Step 3: Feature Scaling (Same code from data pre-processing)

Step 4: Model Training (Logistic Regression)

We trained a **Logistic Regression** model on the training data (X\_train, Y\_train) and used the model to predict the target variable (Y\_test) on the test set.

from sklearn.linear\_model import LogisticRegression

classifier = LogisticRegression(random\_state=0)

classifier.fit(X\_train, Y\_train)

Y\_pred = classifier.predict(X\_test)

Step 5: Model Evaluation

from sklearn.metrics import confusion\_matrix, classification\_report, accuracy\_score

cm = confusion\_matrix(Y\_test, Y\_pred)

**precision recall f1-score support**

0 0.89 0.96 0.92 68

1 0.89 0.75 0.81 32

**accuracy** 0.89 100

**macro avg** 0.89 0.85 0.87 100

**weighted avg** 0.89 0.89 0.89 100

[[65, 3],

[ 8, 24]]

print("Confusion Matrix:")

**Accuracy**: The Logistic Regression model achieved an accuracy of 89%.

print(cm)

print("Classification Report:") Accuracy:

print(classification\_report(Y\_test, Y\_pred))

prediction = accuracy\_score(Y\_test, Y\_pred)

print("Accuracy : ", prediction \* 100, "%")

Lab Summary:

we used **Logistic Regression** to classify user purchases based on their **Age** and **Estimated Salary**, achieving an accuracy of **89%**.

**Lab Eaxm: Midterm exam. Oct 8, 2024 - Customer Purchasing Behaviors.csv**

**Lab 8: Feature Selection. Oct 15, 2024**

we focused on Feature Selection techniques using the Iris dataset to improve classification models. Here’s a summary of the key steps and results:

**Variance Threshold:** We dropped features with zero variance. This step did not eliminate any features in the Iris dataset since all features had some variance.

**Univariate Feature Selection (Chi-square Test):** We used the Chi-square test to select the top 2 and then 3 best features. The most informative features based on the Chi-square test were the petal length and petal width.

**Model-based Feature Selection (Random Forest):** We used a Random Forest classifier to select important features based on feature importance scores. Features like petal length and petal width were identified as the most important.

**Classification Pipeline:** We combined feature selection with a KNN classifier and Random Forest model.

The process included:

Feature selection using Chi-square (top 2 features).

Further feature selection using a Random Forest model.

Classification using KNN.

**Accuracy:** The classification pipeline achieved a high accuracy of **97.78%** on the test set.

**Feature Importance:** The petal length and petal width had the highest importance scores from the Random Forest model, highlighting their significance in classifying the Iris species.

**Lab 9: Feature Selection. Oct 22, 2024**

In this part of the lab, we applied **Bagging** (Bootstrap Aggregating) using **Random Forest** and **Decision Tree Classifiers** on the **Social Network Ads** dataset. The goal was to observe how changing different hyperparameters and models affects the **accuracy** and **performance metrics** of the classifiers.

Decision Tree Classifier (DTC)

This classifier shows a good balance between both classes, but it performs slightly better on predicting class 0 (non-purchased) compared to class 1 (purchased).

**Accuracy: 88%**

Precision, Recall, F1-Score:

For class 0 (not purchased): Precision 0.88, Recall 0.94, F1-Score 0.91

For class 1 (purchased): Precision 0.88, Recall 0.78, F1-Score 0.83

**Model: Decision Tree Classifier (DTC)**

**Criterion: 'entropy'** (Information gain used for splitting)

**Max Depth: 5** (limiting the tree depth to avoid overfitting)

Random Forest performed better than the Decision Tree, with higher overall accuracy and better balance between both classes. The classifier was able to handle the class imbalance more effectively, especially with better recall for class 1.

**Accuracy: 92%**

Precision, Recall, F1-Score:

For class 0 (not purchased): Precision 0.95, Recall 0.92, F1-Score 0.94

For class 1 (purchased): Precision 0.87, Recall 0.92, F1-Score 0.89

**Random Forest Classifier (RF) with 100 Estimators**

**Model**: **Random Forest Classifier (RF)**

**n\_estimators**: 100 (number of decision trees in the forest)

**Max Depth**: **5** (limit tree depth to avoid overfitting)

**Accuracy: 90%**

Precision, Recall, F1-Score:

For class 0 (not purchased): Precision 0.90, Recall 0.95, F1-Score 0.92

For class 1 (purchased): Precision 0.91, Recall 0.81, F1-Score 0.86

**Random Forest Classifier (RF) with 10 Estimators**

Random Forest with shallow trees (max depth = 1) still achieved good accuracy (90%) but with lower recall for class 1 compared to the previous Random Forest model (with deeper trees). The model underperformed slightly in terms of correctly identifying class 1.

**and Depth 1 (Shallow Trees**)

**Model**: **Random Forest Classifier (RF)**

**n\_estimators**: 10 (fewer trees)

**Max Depth**: **1** (shallow trees)

With 50 trees and a max depth of 3, the model shows a **significant improvement in accuracy (94%)**. It performs very well with high precision and recall for class 0 and better recall for class 1 compared to the previous models.

**Accuracy: 94%**

Precision, Recall, F1-Score:

For class 0 (not purchased): Precision 0.97, Recall 0.94, F1-Score 0.96

For class 1 (purchased): Precision 0.88, Recall 0.94, F1-Score 0.91

#### Random Forest Classifier (RF) with 50 Estimators

#### and Max Depth 3

**Model**: **Random Forest Classifier (RF)**

**n\_estimators**: 50 (more trees than before)

**Max Depth**: 3 (more complex trees than depth 1)

#### Random Forest Classifier (RF) with 75 Estimators

While the accuracy remained at 92%, this model showed a **slight decline in precision for class** 0 compared to the 50-tree model, but the recall for class 1 increased slightly.

**Accuracy: 92%**

Precision, Recall, F1-Score:

For class 0 (not purchased): Precision 0.97, Recall 0.90, F1-Score 0.93

For class 1 (purchased): Precision 0.85, Recall 0.95, F1-Score 0.90

#### and Max Depth 3

#### Model: Random Forest Classifier (RF)

#### n\_estimators: 75 (more trees)

#### Max Depth: 3 (same as previous model)

In this part of the lab, we used **AdaBoost** (Adaptive Boosting) with a **Decision Tree classifier** as the base model to predict whether a mushroom is edible or poisonous based on various features.

Step 1: Data Import (**mushrooms.csv)**

**Decision Tree Classifier:** The base estimator for AdaBoost is a Decision Tree with a max depth of 1 (essentially a stump). A Decision Tree of depth 1 is a very simple model, which can only make binary decisions based on one feature.

**AdaBoost Classifier**: AdaBoost is applied by passing the DecisionTreeClassifier as the estimator. The main idea behind AdaBoost is that it trains multiple models sequentially, where each subsequent model focuses on the **misclassified data points** from the previous model.

n\_estimators = 400: 400 rounds of boosting are performed. This means 400 weak models (decision tree stumps) are created, and they are combined to form the final strong model.

learning\_rate = 1: This controls how much weight is given to each model in the ensemble. A learning rate of 1 means each model will contribute equally to the final prediction.

Step 2: **Label Encoding**

Step 3: Splitting the Dataset

Step 4: AdaBoost with Decision Tree as Base Classifier

# Create Decision Tree classifier object (depth=1, as weak learner)

model = DecisionTreeClassifier(criterion='entropy', max\_depth=1)

# Create AdaBoost classifier object with Decision Tree as base estimator

AdaBoost = AdaBoostClassifier(estimator=model, n\_estimators=400, learning\_rate=1)

# Train the AdaBoost

model boostmodel = AdaBoost.fit(X\_train, y\_train)

# Predict the response for the test dataset

y\_pred = boostmodel.predict(X\_test)

Step 5: Evaluation

**Accuracy**: The model achieved **100% accuracy** on the test set. This means that the AdaBoost model correctly predicted all of the mushroom classifications (whether they are edible or poisonous) for the test data.

Lab Summary:

**Bagging** (Random Forest) effectively improved accuracy by combining multiple decision trees trained on random subsets of the data. It provided better accuracy than a single decision tree model.

**Boosting** (AdaBoost) provided superior performance, achieving 100% accuracy by correcting errors from previous learners, highlighting its strength in reducing bias and improving model performance, especially when there are misclassified examples.

**Lab 10: K-Nearest Neighbors (k-NN) Classifier with Hyperparameter Tuning. Nov 5, 2024**

The objective of this lab is to implement and evaluate the performance of the k-Nearest Neighbors (k-NN) algorithm on the **Social Network Ads dataset** and tune the model by varying the value of k, and applying feature scaling to improve accuracy.

Step 1: Data Import (same as the Lab 9 Part A **Social Network Ads dataset)**

Step 2: Data Pre-processing

Split data into training and test sets (25% test data).

Train the k-NN classifier with k=5 and evaluate accuracy.

**Output**: Accuracy = **0.82**.

Experiment with k=7.

**Output**: Accuracy = **0.83**.

Change test size to 20% and re-evaluate with k=7.

**Output**: Accuracy = **0.83**.

Step 3: Split the Data into Features and Target

Step 4: Model k-NN

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.25, random\_state=42)

k = 5

knn = KNeighborsClassifier(n\_neighbors=k)

knn.fit(X\_train, y\_train)

y\_pred\_knn = knn.predict(X\_test)

accuracy\_knn = accuracy\_score(y\_test, y\_pred\_knn)

classification\_rep\_knn = classification\_report(y\_test, y\_pred\_knn)

print(f"Accuracy (k-NN): {accuracy\_knn}")

print(classification\_rep\_knn)

Step 5: Feature Scaling Using StandardScaler

**Output**: Optimal k = 8, Accuracy = **95.00%**.

**Train the model using the** Minkowski distance **with p=2 (Euclidean distance) Output: Optimal k = 8, Accuracy = 95.00%.**

The results indicate that **feature scaling** and **optimal choice of** k significantly improve the model's performance. A value of **k=8** with feature scaling and the **Euclidean distance** metric resulted in the best classification accuracy (95%).

|  |  |
| --- | --- |
| **Model Configuration** | **Accuracy (%)** |
| k-NN with k=5 | **82** |
| k-NN with k=7 | **83** |
| k-NN with k=7 (80/20 Split) | **83** |
| k-NN with optimal k=8 (Scaling) | **95** |
| k-NN with optimal k=8 (Minkowski Metric) | **95** |

Lab Summary:

By experimenting with the **k-NN classifier**, we learned how the choice of k, feature scaling, and distance metrics can influence model performance. This process highlights the importance of **hyperparameter tuning** and **preprocessing** in achieving high classification accuracy.

The final model with **k=8**, after scaling the features, showed the best performance, achieving **95% accuracy**, indicating that k-NN can be a highly effective classifier when fine-tuned appropriately.