University of Tehran
School of Mechanical Engineering

Artificial Intelligence Assignment No.2

Author
Emil Keshishian

Course Instructor
Dr. Masoud Shariat Panahi

May 2024

Contents

List of Figures

List of Tables

List of Listings

1 Regression

All data analysis and manipulation tasks were performed using the *Python* programming language and the relevant libraries.

1.1 Raw Data Analysis

1.1.1 DataFrame Info List

The following code snippet (1) was used for reading the CSV file containing the dataset.

Snippet 1: Reading the Data

```
data = pd.read_csv("CarPrice.csv", index_col=0, header=0)
print(data.info())
```

which generated the following info list 2:

Snippet 2: DataFrame info output

```
Int64Index: 205 entries, 1 to 205
Data columns (total 25 columns):
#   Column            Non-Null Count  Dtype
---  ------            --------------  -----
0   symboling         205 non-null    int64
1   CarName           205 non-null    object
2   fueltype          205 non-null    object
3   aspiration        205 non-null    object
4   doornumber        205 non-null    object
5   carbody           205 non-null    object
6   drivewheel        205 non-null    object
7   enginelocation    205 non-null    object
8   wheelbase         205 non-null    float64
9   carlength         205 non-null    float64
10  carwidth          205 non-null    float64
11  carheight         205 non-null    float64
12  curbweight        205 non-null    int64
13  enginetype        205 non-null    object
14  cylindernumber    205 non-null    object
15  enginesize        205 non-null    int64
16  fuelsystem        205 non-null    object
17  boreratio         205 non-null    float64
18  stroke            205 non-null    float64
19  compressionratio  205 non-null    float64
20  horsepower        205 non-null    int64
21  peakrpm           205 non-null    int64
22  citympg           205 non-null    int64
23  highwaympg        205 non-null    int64
24  price             205 non-null    float64
dtypes: float64(8), int64(7), object(10)
memory usage: 41.6+ KB
```

As visible in the info list above, there are numerical as well as categorical data available in the dataframe.

1.1.2 DataFrame Statistical Analysis

The maximum, minimum and standard variation values are calculated for the prices' column with the following code snippet (3).

Snippet 3: Prices' Min, Max and SV

```
print(data.describe())
print("Highest price is: ", data.get("price").max())
print("Lowest price is: ", data.get("price").min())
print("The standard diviation of the Prices is: ", data.get("price").std())
```

The computed values are presented in table 1.

Table 1: Car price statistical values

Label	Value
Highest Price	45400.0
Minimum Value	5118.0
Standard Diviation	7988.8523

1.1.3 Correlation Matrix

In order to plot the correlation matrix for the given dataset, the categorical data columns should be transformed into numerical ones, for this purpose both one-hot labeling and indexing are utilized. In table 2 the numeration method of each categorical column is specified.

Table 2: parameter numeration methods

Label	Numeration Type
car name	indexing
fueltype	one-hot
aspiration	indexing
doornumber	indexing
carbody	one-hot
drivewheel	indexing
enginelocation	indexing
enginetype	one-hot
cylindernumber	converted to int
fuelsystem	one-hot

The rationale behind choosing one-hot or ordered indexing behind each parameter is just a common knowledge about car specifications and their prices. For example two-door cars are generally more expensive hence they could be indexed practically.

For converting car names into numerals only the car brand was used as speculated by the author the brand of the vehicle is more tightly connected to its price hence for example in *alfa-romero giulia* only *alfa-romero* is used for indexing. One-hot labeling could have been also utilized but it would have increased the size of the data set by 22 (this number will be explained shortly) extra columns which is undesired. in order to convert the *CarName* column into numeral values code snippet 4 is used were a list of all car brands included in the dataset is created by hand and the *CarName* column is first converted to the brand names and then each brand name is associated to its index value in the brand name list.

Snippet 4: Car names Processing

```python
vehicle_names = [
    "alfa-romero", "audi", "bmw", "chevrolet", "dodge", "honda", "isuzu", "jaguar", "mazda", "buick", "mercury", "mitsubishi", "nissan", "peugeot", "plymouth", "porsche", "renault", "saab", "subaru", "toyota", "volkswagen", "vw", "volvo"
]

# Seperating car brands from the complete name string
for idx, name in enumerate(data["CarName"]):
    for bname in vehicle_names:
        if name.lower().find(bname) != -1:
            data["CarName"][idx+1] = bname
            break

# Similarity double-checking and correction
for idx, name in enumerate(data["CarName"]):
    if name.lower() not in vehicle_names:
        for bname in vehicle_names:
            if sc(a=name.lower()[0:name.find(" ")], b=bname).ratio() > 0.7:
                data["CarName"][idx + 1] = bname
                #print(name, idx)
                break

# Final Check
for idx, name in enumerate(data["CarName"]):
    if name.lower() not in vehicle_names:
        PassFlag = True
        print(name)

for idx, name in enumerate(data["CarName"]):
    for idx2, bname in enumerate(vehicle_names):
        if name.lower().find(bname) != -1:
            data["CarName"][idx+1] = idx2

print(data["CarName"])
```

For speeding up the one-hot labeling of the specified columns the **onehot** function was written, refer to snippet 5

Snippet 5: Onehot function

```python
def onehot(data, key):
    list = data[key].unique()
```

```
    for i, k in enumerate(list):
        data.insert(data.columns.get_loc(key), key + ' ' + str(i) + ' ' + str(k), np.zeros(len(data[key])))
    for i, k in enumerate(list):
        for idx, val in enumerate(data[key]):
            if val == k:
                data[key + ' ' + str(i) + ' ' + str(k)][idx] = 1
    data.drop(key, axis=1, inplace=True)
```

Using the above function, processing the rest of the columns became trivial. Code snippet 6 is the code for converting the remaining columns.

Snippet 6: Converting the columns

```
data['CarName'] = data['CarName'].astype(int)
onehot(data, 'fueltype')
print(data[data.columns[1:4]][60:80])
data["aspiration"].replace({'std':0, 'turbo':1}, inplace=True)
data["doornumber"].replace({'two':1, 'four':0}, inplace=True)
onehot(data, 'carbody')
print(data[data.columns[5:10]][60:80])
data["drivewheel"].replace({'rwd':2, 'fwd':0, '4wd':1}, inplace=True)
data["enginelocation"].replace({'front':0, 'rear':1}, inplace=True)
onehot(data, 'enginetype')
data["cylindernumber"].replace({'two':2, 'three':3, 'four':4, 'five':5, 'six':6, 'eight':8, 'twelve':12}, inplace=True)
onehot(data, 'fuelsystem')
```

A fresh overview of the data frame shows the newly defined columns for one-hot labeling.

Snippet 7: Data frame info after the non-numerical data conversion

```
Data columns (total 43 columns):
#   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
0   symboling              205 non-null    int64
1   CarName                205 non-null    int32
2   fueltype 0 gas         205 non-null    float64
3   fueltype 1 diesel      205 non-null    float64
4   aspiration             205 non-null    int64
5   doornumber             205 non-null    int64
6   carbody 0 convertible  205 non-null    float64
7   carbody 1 hatchback    205 non-null    float64
8   carbody 2 sedan        205 non-null    float64
9   carbody 3 wagon        205 non-null    float64
10  carbody 4 hardtop      205 non-null    float64
11  drivewheel             205 non-null    int64
12  enginelocation         205 non-null    int64
13  wheelbase              205 non-null    float64
14  carlength              205 non-null    float64
15  carwidth               205 non-null    float64
16  carheight              205 non-null    float64
17  curbweight             205 non-null    int64
18  enginetype 0 dohc      205 non-null    float64
19  enginetype 1 ohcv      205 non-null    float64
20  enginetype 2 ohc       205 non-null    float64
21  enginetype 3 l         205 non-null    float64
```

```
22  enginetype 4 rotor     205 non-null    float64
23  enginetype 5 ohcf      205 non-null    float64
24  enginetype 6 dohcv     205 non-null    float64
25  cylindernumber         205 non-null    int64
26  enginesize             205 non-null    int64
27  fuelsystem 0 mpfi      205 non-null    float64
28  fuelsystem 1 2bbl      205 non-null    float64
29  fuelsystem 2 mfi       205 non-null    float64
30  fuelsystem 3 1bbl      205 non-null    float64
31  fuelsystem 4 spfi      205 non-null    float64
32  fuelsystem 5 4bbl      205 non-null    float64
33  fuelsystem 6 idi       205 non-null    float64
34  fuelsystem 7 spdi      205 non-null    float64
35  boreratio              205 non-null    float64
36  stroke                 205 non-null    float64
37  compressionratio       205 non-null    float64
38  horsepower             205 non-null    int64
39  peakrpm                205 non-null    int64
40  citympg                205 non-null    int64
41  highwaympg             205 non-null    int64
42  price                  205 non-null    float64
```

With the non-numerical features converted to numerical, the correlation matrix is computed and plotted using seaborns heatmap plotter function in snippet 8. The correlation matrix is visualized in fig. 1.

Snippet 8: Correlation matrix generation and plotting

```python
plt.figure(figsize=(10, 10))
sns.heatmap(data.corr(), annot=False, fmt='.2f', cbar=True)
#plt.matshow(data.corr())
plt.savefig('correlation.png')
plt.show()
```

As evident by a qualitative assemsment of the visualized correlation matrix, the price of a car is mostly dependent on its horsepower, bore ratio, engine size and curb weight.

1.1.4 Data Preprocessing

By analysing the correlation matrix it is evident that some features like symboling, citympg and highwaympg are irrelevant to price, hence it's possible to remove them (care is taken in removing data).

1.1.5 Normalization

One of the main steps in data pre-processing is data standardazation and normalization. This is not an optional step as most regression algorithms are built upon assumptions regarding normalized and standard data, hence a normalization step was carried out utilizing sklearn's preprocessing normalize module. The Data has been normalized along the sample axes meaning each column is normalized relative to itself. The normalization code is available in snippet **??**. The results of different models trained on not standardized and and not normalized data are compared to the processed data in table 5.

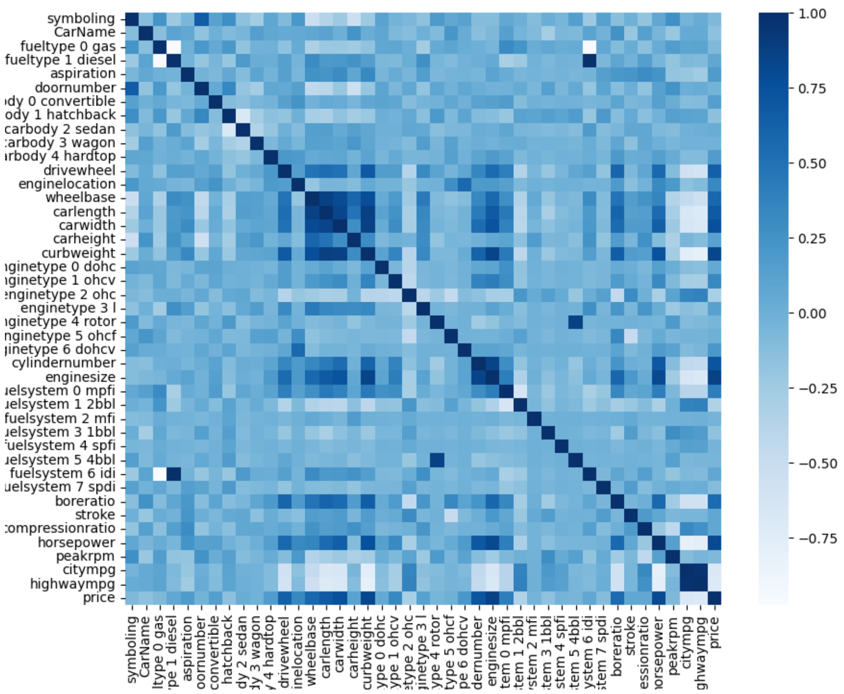


Figure 1: The correlation matrix

```
scaler = StandardScaler()
d = scaler.fit_transform(data)
data = pd.DataFrame(data=d, columns=data.columns)

print("\n Standardized Data:")
print(data.head(10))

d = normalize(data, norm="l1", axis=0)
data = pd.DataFrame(data=d, columns=data.columns)
```

1.1.6 Model Selection, Training and Evaluation

First the dataset is devided into training and test portions with **Sklearn**'s *train-test-split* function. Then using the code available in snippet 10 enginesize-price and horsepower-price joint plots are drawn and illustrated in figures 2-a and 2- respectively.

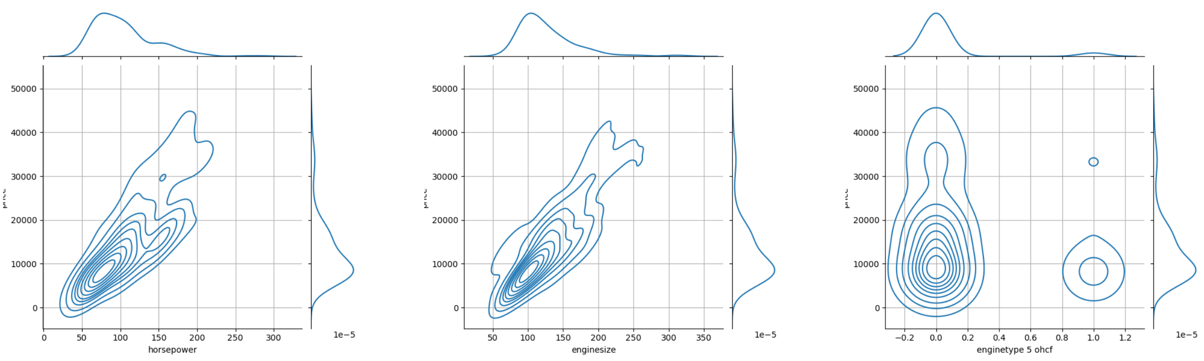
Snippet 9: Data splitting

```
x_train, x_test, y_train, y_test = train_test_split(data[data.columns[:-1]],
↪  data['price'], test_size=0.3, random_state=42)
```

As evident from the joint plots, both features are suitable for predicting the price. An example of a joint plot indicating a poorly related feature to the price is available in figure , which was plotted for the worst feature selected by SelectKBest, enginetype 5 ohcf.

Snippet 10: Engine power - engine size joint plot

```
plt.figure(figsize=(10,10))
sns.jointplot(x=data['horsepower'], y=data['price'], kind='kde', data=data)
```



(a) engine power - price joint plot (b) engine size - price joint plot (c) engine type - price joint plot

Figure 2: Joint plots

```
plt.grid()
plt.savefig('hp-price_jointplot.png')
plt.show()

plt.figure(figsize=(10,10))
sns.jointplot(x=data['enginesize'], y=data['price'], kind='kde', data=data)
plt.grid()
plt.savefig('enginesize-price_jointplot.png')
plt.show()
```

For the next step the SelectKbest function from the **Scikitlearn** was used again, and the top 10 features were selected, which were drivewheel, wheelbase, carlength, carwidth, curbweight, cylindernumber, enginesize, horsepower, citympg, highwaympg. The feature selection code is available in snippet 11

Snippet 11: Top ten features selected

```
selector = SelectKBest(f_regression, k=10)
x_new = selector.fit_transform(x, y)
print(x.columns[selector.get_support()])

out:
Index(['drivewheel', 'wheelbase', 'carlength', 'carwidth', 'curbweight',
       'cylindernumber', 'enginesize', 'horsepower', 'citympg', 'highwaympg'],
      dtype='object')
```

For training three regression models are used, linear regression, lasso regression, ridge regression and SVM. All of the mentioned models have been imported from sklearn. For the linear regressor all default parameters have been used. In Lasso and Ridge regressors, the alpha value is a non-negative floating-point number that scales the L1 penalty term, thereby influencing the level of regularization applied. In support vector regression kernel Specifies the type of kernel to be used in the algorithm. The default is 'rbf', which stands for Radial Basis Function, a common choice for non-linear data. C is the regularization parameter. A higher value of C indicates a lower tolerance for errors and leads to a narrower margin, It must be a strictly positive float. The strength of the regularization is inversely proportional to C. epsilon Defines the epsilon-tube within which no penalty is associated in the training loss function with points predicted within a distance epsilon from the actual value, It must be non-negative.

Two measures of performance were considered for model evaluation, **RMSE** and R^2. RMSE is a standard deviation of the residuals (prediction errors). Residuals are the differences between the observed values and the predicted values by the model. RMSE measures how spread out these residuals are; in other words, it tells you how concentrated the data is around the line of best fit. R^2 is a statistical measure that determines the proportion of variance in the dependent variable that can be explained by the independent variable(s). It is a value between 0 and 1 and is often used to indicate the goodness of fit of a model. An R^2 of 0 suggests that the model does not explain any of the variability of the response data around its mean, while an R^2 of 1 indicates that the model explains all the variability of the response data around its mean.

The RMSE values of all models available in table 3 For all models the coefficient of determination is calculated for both test and train data and are presented in table 5.

Table 3: RMSE score comparison of all models

Model	**Linear**	**Lasso**	**Ridge**	**SVR**
RMSE	0.23859587	0.247908396	0.238589455	0.289251269

Table 4: Normalized regression models' R^2 score comparison

R^2 **Score**	**Linear**	**Lasso**	**Ridge**	**Support Vector**
Training	0.8505	0	0.7581	-2.3041
Test	0.7812	-0.0027	0.7223	-2.1787

Table 5: Standardized regression models' R^2 score compariso

R^2 **Score**	**Linear**	**Lasso**	**Ridge**	**Support Vector**
Training	0.8505	0.8337	0.8505	0.9266
Test	0.7812	0.7727	0.7812	0.7348

It is clear that unlike standardization, normalization is not a suitable preprocessing step hence only standardization has been performed.

2 Classification

2.1 Data Preparation

The first step in data preparation is having an overview of the entire data frame. As visible in data info 12 there all columns except for Outcome which is the label column, have empty cells which means a cleanup of the empty values is crucial.

Snippet 12: DataFrame info output

```
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
^^I#   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
^^I0   Pregnancies               635 non-null    float64
^^I1   Glucose                   654 non-null    float64
^^I2   BloodPressure             680 non-null    float64
^^I3   SkinThickness             624 non-null    float64
^^I4   Insulin                   680 non-null    float64
^^I5   BMI                       684 non-null    float64
^^I6   DiabetesPedigreeFunction  590 non-null    float64
^^I7   Age                       655 non-null    float64
^^I8   Outcome                   768 non-null    int64
dtypes: float64(8), int64(1)
memory usage: 54.1 KB
```

The number and percentage of missing values of each column is computed using the code snippet 13

Snippet 13: Empty value count and percentage calculation

```
nansums = []
data_length = len(data)
nanpercentages = []
for d in data.columns:
^^Inansums.append(data[d].isna().sum())
^^Inanpercentages.append(nansums[-1]/data_length * 100)

print("\nNumber of Nan Values for each column: ")
for i in range(len(data.columns)):
^^Iprint(str(data.columns[i]) + ": " + str(nansums[i]))

print("\nPercentage of Nan Values for each column: ")
for i in range(len(data.columns)):
^^Iprint(str(data.columns[i]) + ": " + str(nanpercentages[i]))
```

To solve the issue with empty values, two methods were tested, data imputation using the mean valuse of each column and dropping all samples containing an empty value. Dropping the samples reduced the data size to the point were training scores fell drastically. Hence imputation was performed.

Snippet 14: Empty values removal

```
# imputation
for c in data.columns:
```

```python
data[c] = data[c].fillna(data[c].mean())
# Dropping Samples
data.dropna(axis=0)
```

Before plotting the correlation matrix of the features standardization and normalization of the data was performed and performances of the models were compared against the unprocessed data. The results of the comparison is presented after the model definition section in table.

Similar to the correlation matrix in the previous section, the correlation matrix of the features are plotted in figure 3.

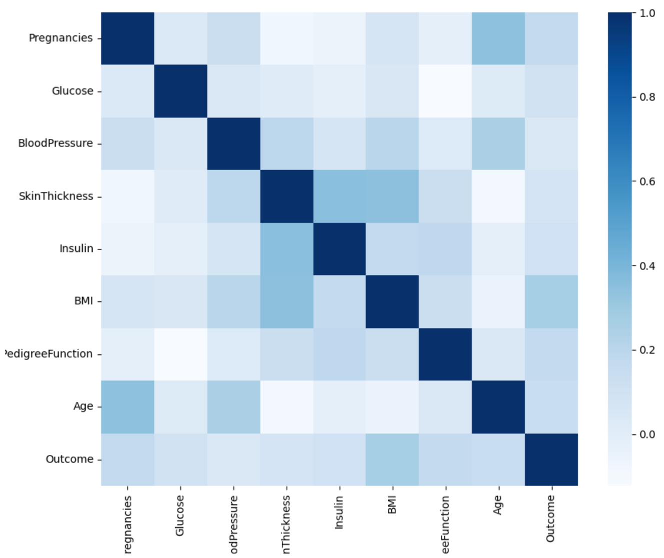
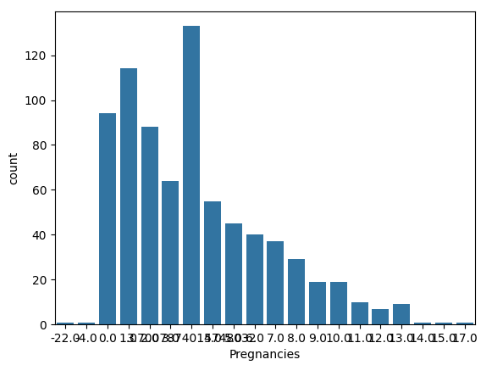
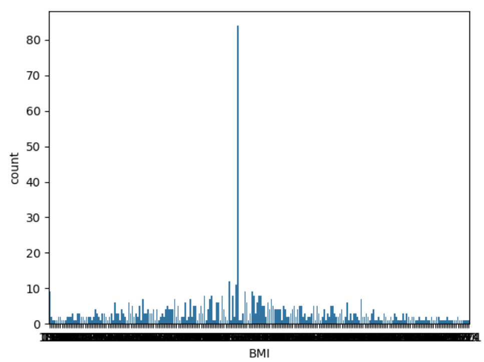


Figure 3: Correlation matrix of the categorical data

As indicated by the correlation matrix Pregnancies and BMI features are correlated with outcome more than the other features. The occurance plots of these features are available in plots and a.

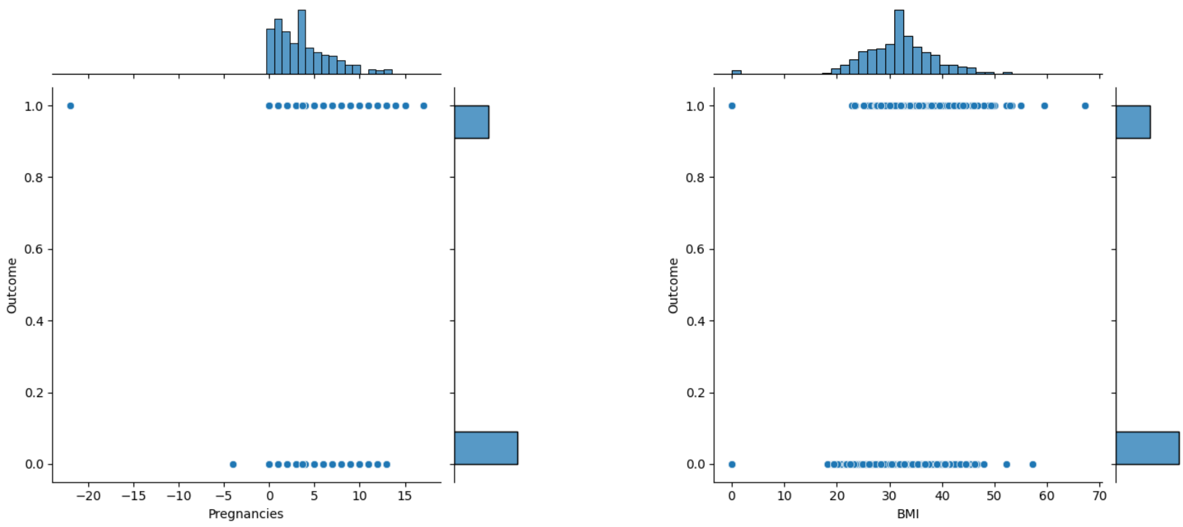




(a) number of unique Pregnancies occurrences (b) number of unique BMI value occurrences

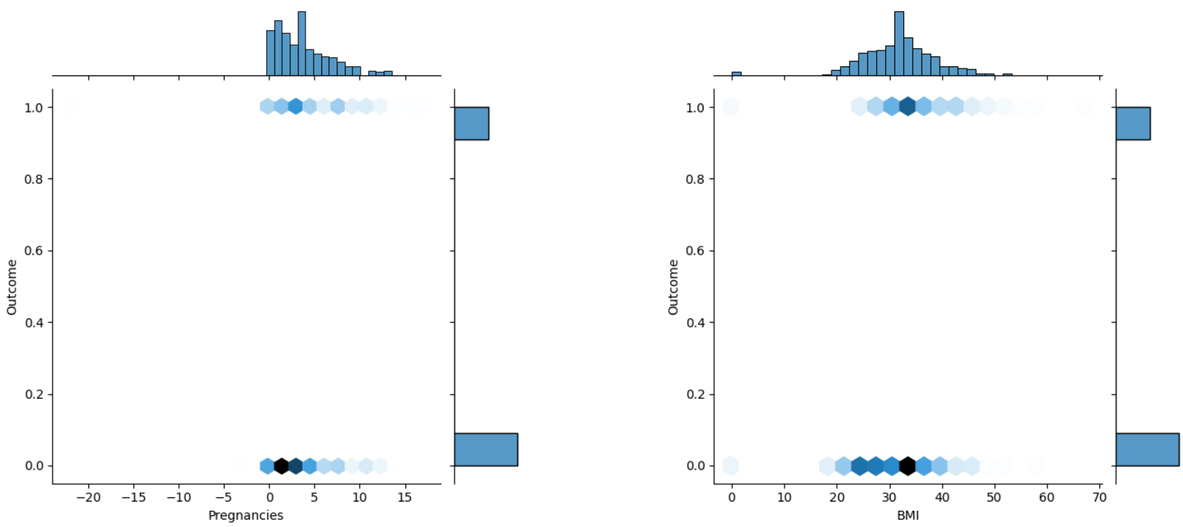
Figure 4: occurance plots

More so the joint plots of both features were also plotted.



(a) Pregnancies-Outcome (b) BMI-Outcome

Figure 5: Scatter joint plots



(a) Pregnancies-Outcome (b) BMI-Outcome

Figure 6: Hex joint plots

2.2 Model Initialization and Training

In this section the process of creating and training the following Classifiers is ellaborated:

- Logistic Regression
- K-Nearest Neighbors
- Decision Tree
- Random Forest
- Support Vector Machine

The above mentioned models are created and trained with the code available in snippet

Snippet 15: Classification models Initialization and training

```
logistic = LogisticRegression(random_state=0, max_iter=10000).fit(x_train, y_train)
lr_train_score = logistic.score(x_train, y_train)
lr_score = logistic.score(x_test, y_test)

KNN = KNeighborsClassifier(n_neighbors=10)
KNN.fit(x_train, y_train)
knn_train_score = KNN.score(x_train, y_train)
knn_score = KNN.score(x_test, y_test)

DT = DecisionTreeClassifier(random_state=0)
DT.fit(x_train, y_train)
dt_train_score = DT.score(x_train, y_train)
dt_score = DT.score(x_test, y_test)

RF = RandomForestClassifier()
RF.fit(x_train, y_train)
rf_train_score = RF.score(x_train, y_train)
rf_score = RF.score(x_test, y_test)
```

The Logistic Regression model is initialized with a random state of 0 and maxiter set to 10000 random state Controls the shuffling applied to the data before applying the split Setting, a random state value ensures reproducibility. Maxiter is the maximum number of iterations taken for the solvers to converge, The model is then trained on the xtrain and ytrain datasets (All Models are trained on xtrain and ytrain except for the last section were a new validation set is defined) The models performance is evaluated on both the training set lrtrainscore and the test set lrscore with the scores printed to the console. The Decision Tree Classifier is created and trained with a random state of 0. The Random Forest Classifier is instantiated with default parameters. The support vector classifier was also setup using sklearn's SVM module with default parameters.

For the parameters of all of the models a grid search optimization algorithm was applied using the **GridSearchCV** module from sklearn which given a collection of parameters and the values that need to be tested, performs the optimization and returns the best combination of the given parameters, the implementation of this method is available in snippet 16.

Snippet 16: GridSearch algorithm implementation

```
^^Iparameters = {
'fit_intercept':[True, False],
'C':[1,10, 20, 50, 100],
'solver':['lbfgs', 'liblinear', 'newton-cg', 'newton-cholesky', 'sag', 'saga']}
loreg = LogisticRegression(max_iter=10000, solver='saga', tol=0.1)
gridsearch = GridSearchCV(loreg, parameters, cv=5).fit(x_train, y_train)
print(gridsearch.best_params_)
#Result: {'C': 10, 'fit_intercept': True, 'penalty': 'l2', 'solver': 'newton-cholesky'}

# KNN
gridsearch = GridSearchCV(KNeighborsClassifier(), {'n_neighbors':[2,5,10,15,20,30]}, cv=5).fit(x_train, y_train)
```

```
print(gridsearch.best_params_)
# Result: {'n_neighbors': 20}

# Decision Tree
parameters = {
    'criterion':['gini', 'entropy', 'log_loss'],
    'splitter':['best', 'random'],
    'max_depth':[2,4,6,8,10,12],
    'min_samples_split':[2,4,6,8,10,12]}

gridsearch = GridSearchCV(DecisionTreeClassifier(), parameters,
↪  cv=5).fit(x_train, y_train)
print(gridsearch.best_params_)
# Result: {'criterion': 'gini', 'max_depth': 2, 'min_samples_split': 2,
↪  'splitter': 'best'}

# Decision Tree
parameters = {
    'n_estimators':[100,200,300,400],
    'criterion':['gini', 'entropy', 'log_loss'],
    'max_depth':[2,4,6,8],
    'min_samples_split':[2,4,6,8,10,12]}

# Random Forest
gridsearch = GridSearchCV(RandomForestClassifier(), parameters,
↪  cv=5).fit(x_train, y_train)
print(gridsearch.best_params_)
# Result: {'criterion': 'gini', 'max_depth': 8, 'min_samples_split': 6,
↪  'n_estimators': 300}

# SVM
parameters = {'kernel':('linear', 'rbf'), 'C':[1, 10]}
SVM = SVC()
gridsearch = GridSearchCV(SVM, parameters, cv=5).fit(x_train, y_train)
print(gridsearch.best_params_)
# Result: {'C': 1, 'kernel': 'linear'}
```

Each models training and test scores provide insights into their performance and generalization capabilities Higher scores on the test set indicate better model generalization. All models before preprocessing and postprocessing performances are compared in table 6. The training and test scores of all models before and after parameter optimization are compared and are available in table 7.

Table 6: Pre- and post-processing performance on test data comparison

Model	No Processing	Normalization	Standardization	Norm and Std
Logistic Regression	0.7467	0.6428	0.7337	0.7142
KNN	0.6688	0.6883	0.6818	0.6493
Decision Tree	0.6233	0.6103	0.6298	0.7337
Random Forest	0.7597	0.6493	0.7402	0.7662
SVM	0.7207	0.6623	0.6753	0.6558

As evident in table 6, there's no single great solution for all models and depending on the model the preprocessing method and weather there's need for any or not could be decided. But if we were to chose the highest scoring combination it would be the random forest with both standardization and normalization (later a problem called overfitting will be addressed).

Table 7: Optimization effect on the performance of the models

	Default Values		**Optimized**		**change(%)**	
Model	Train	Test	Train	Test	Train	Test
LR	0.7785	0.7402	0.7817	0.7402	100.41	100
KNN	0.7654	0.6688	0.7589	0.6818	99.14	101.94
DT	1.0	0.6233	0.7573	0.7012	75.73	112.5
RF	1.0	0.7402	0.912	0.7272	91.2	98.24
SVM	0.7557	0.7207	0.7687	0.7467	101.72	103.6

Similar to preprocessing there's no single solution for all models. For example in the case of random forest the optimization has had adverse effects (without considering overfitting) but on the other hand Decision Tree has experienced "great" improvement. But a general consideration yields a positive outcome for the optimization.

The correlation matrices for each model before and after optimization are available from figures 7-a to reffig:clscorrmats-j.

It's visible from the data that the Random Forest and KNN models have encountered overfitting, hence a new validation split has been performed and the results are regenerated and presented in table 17, the code is also available in snippet 8.

Snippet 17: Run with validation data

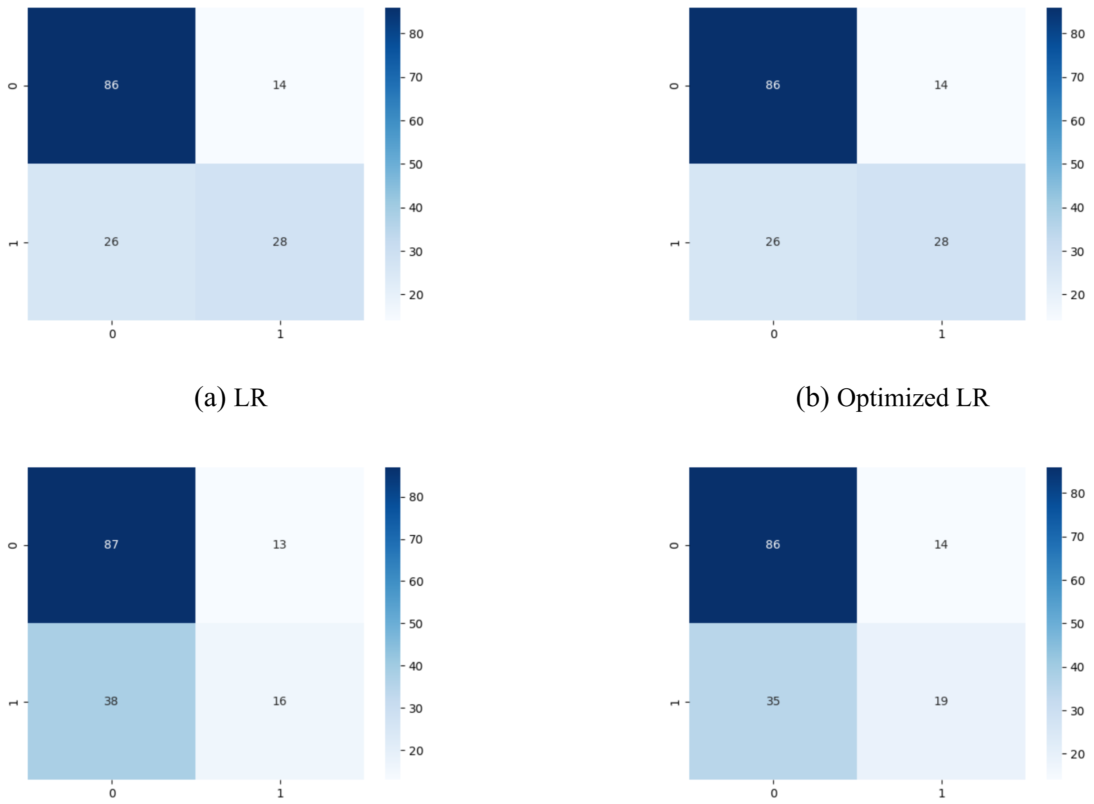
```python
x_train, x_test, y_train, y_test = train_test_split(x, y, stratify=y,
    test_size=0.15, random_state=42)
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train,
    test_size=0.21428, random_state=42)

KNN.fit(x_train, y_train)
valid_knn_score = KNN.score(x_val, y_val)
print('\nKNN validation score: ')
print(valid_knn_score)
print(str(valid_knn_score/new_knn_score * 100) + '% score change with validation
    data')

RF.fit(x_train, y_train)
valid_rf_score = RF.score(x_val, y_val)
print('\nRF validation score: ')
print(valid_rf_score)
print(str(valid_rf_score/new_rf_score * 100) + '% score change with validation
    data')
```

Both Random Forest and Decision Tree are powerful algorithms used for regression and classification tasks. there are two forms of problems called Bias and Variance problems. Bias Arises from simplifications made in the model's assumptions. High bias models may not fit the

Table 8: Validation data training scores

Model	Score	Change(%)
KNN	0.7285	106.85
RF	0.7785	107.05

data well (underfitting). Variance Reflects the model's sensitivity to variations in the training data. High variance models overfit the data.



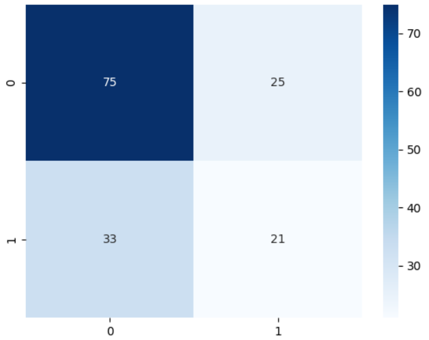
(a) LR (b) Optimized LR

(c) KNN (d) Optimized KNN

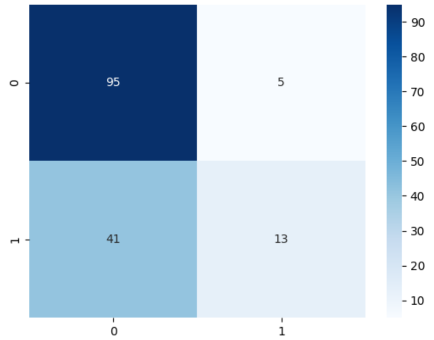
Figure 7: Optimized and unoptimized correlation matrices for all models

"Science is knowledge which we understand so well that we can teach it to a computer. Everything else is art."

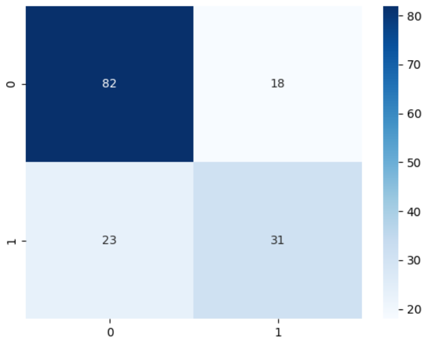
-Donald Knuth



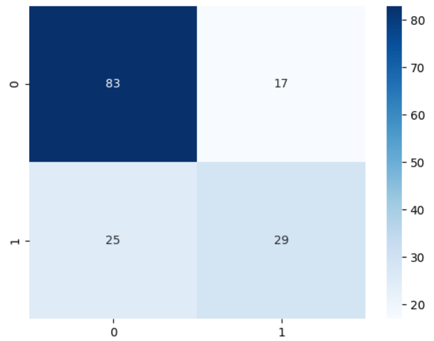
(e) DT



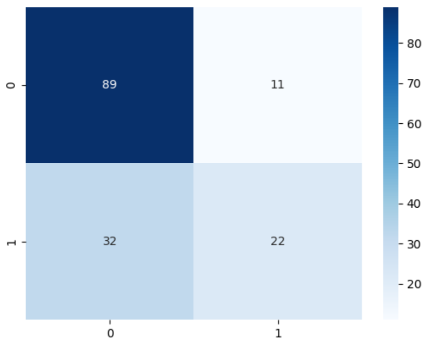
(f) Optimized DT



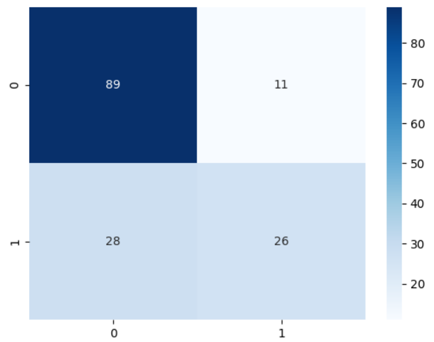
(g) RF



(h) Optimized RF



(i) SVM



(j) Optimized SVM

Figure 7: Optimized and unoptimized correlation matrices for all models