T.C. MALTEPE UNIVERSITY

Department of Computer Engineering

SE 416 Data Mining Term Project

The Subject of the Project: Diabetes

PREPARED BY :

Ertuğrul BAYRAKTAR

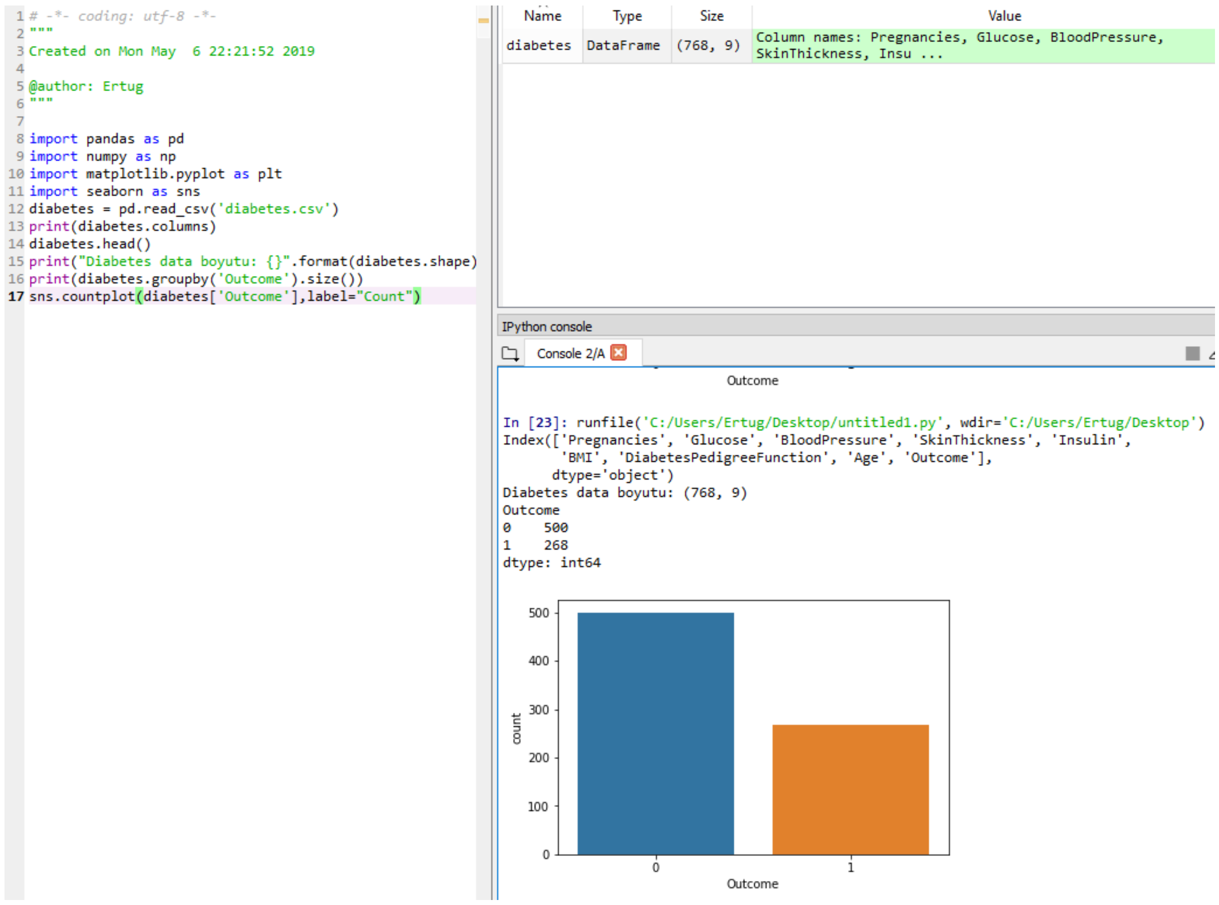
Yrd. Doç. Dr. Volkan TUNALI

1. INTRODUCTION

1.1. Definition of the project

Currently, one in seven adults in the US has diabetes. However, by 2050 this ratio may increase to one third. Our goal is to learn how to use Machine Learning to help us predict Diabetes.

2. DATA



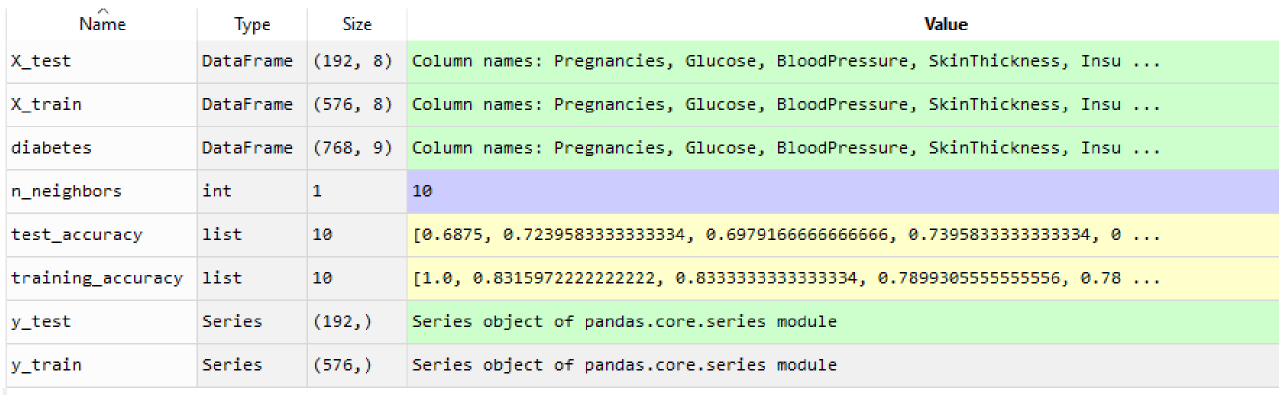
```python
# -*- coding: utf-8 -*-
"""
Created on Mon May  6 22:21:52 2019

@author: Ertug
"""

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
diabetes = pd.read_csv('diabetes.csv')
print(diabetes.columns)
diabetes.head()
print("Diabetes data boyutu: {}".format(diabetes.shape)
print(diabetes.groupby('Outcome').size())
sns.countplot(diabetes['Outcome'],label="Count")
```

Name	Type	Size	Value
diabetes	DataFrame	(768, 9)	Column names: Pregnancies, Glucose, BloodPressure, SkinThickness, Insu ...

```
In [23]: runfile('C:/Users/Ertug/Desktop/untitled1.py', wdir='C:/Users/Ertug/Desktop')
Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin',
       'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome'],
      dtype='object')
Diabetes data boyutu: (768, 9)
Outcome
0    500
1    268
dtype: int64
```

Figure 2.1.

The diabetes data set consists of 768 data points, with 9 features each.
Dimension of diabetes data: (768, 9).
“Outcome” is the feature we are going to predict, 0 means No diabetes, 1 means diabetes. Of these 768 data points, 500 are labeled as 0 and 268 as 1.



Name	Type	Size	Value
X_test	DataFrame	(192, 8)	Column names: Pregnancies, Glucose, BloodPressure, SkinThickness, Insu ...
X_train	DataFrame	(576, 8)	Column names: Pregnancies, Glucose, BloodPressure, SkinThickness, Insu ...
diabetes	DataFrame	(768, 9)	Column names: Pregnancies, Glucose, BloodPressure, SkinThickness, Insu ...
n_neighbors	int	1	10
test_accuracy	list	10	[0.6875, 0.7239583333333334, 0.6979166666666666, 0.7395833333333334, 0 ...
training_accuracy	list	10	[1.0, 0.8315972222222222, 0.8333333333333334, 0.7899305555555556, 0.78 ...
y_test	Series	(192,)	Series object of pandas.core.series module
y_train	Series	(576,)	Series object of pandas.core.series module

Figure 2.2.

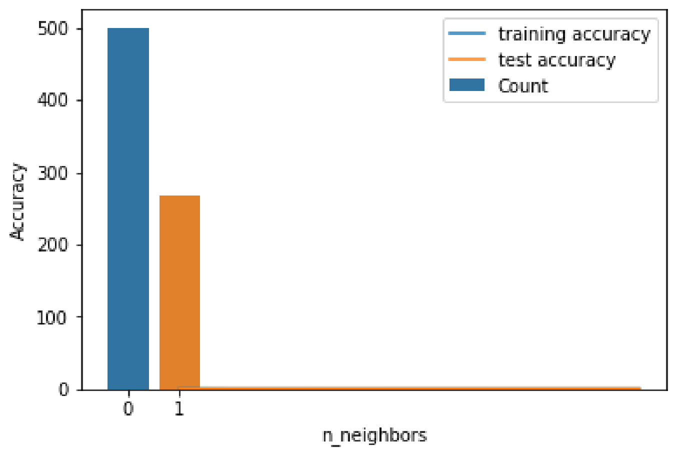
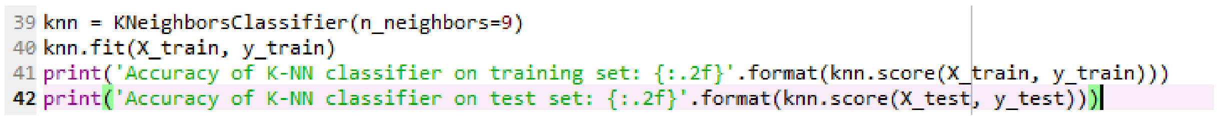


Figure 2.3.

3. Data Mining Algortihms

3.1. k-Nearest Neighbors



```
knn = KNeighborsClassifier(n_neighbors=9)
knn.fit(X_train, y_train)
print('Accuracy of K-NN classifier on training set: {:.2f}'.format(knn.score(X_train, y_train)))
print('Accuracy of K-NN classifier on test set: {:.2f}'.format(knn.score(X_test, y_test)))
```

The k-NN algorithm is arguably the simplest machine learning algorithm. Building the model consists only of storing the training data set. To make a prediction for a new data point, the algorithm finds the closest data points in the training data set—its "nearest neighbors."

First, Let's investigate whether we can confirm the connection between model complexity and accuracy:

```
Accuracy of K-NN classifier on training set: 0.79
Accuracy of K-NN classifier on test set: 0.78
```

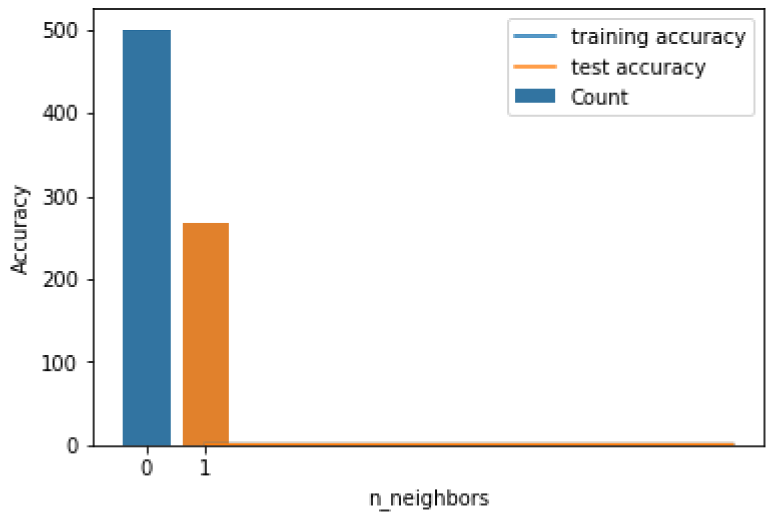
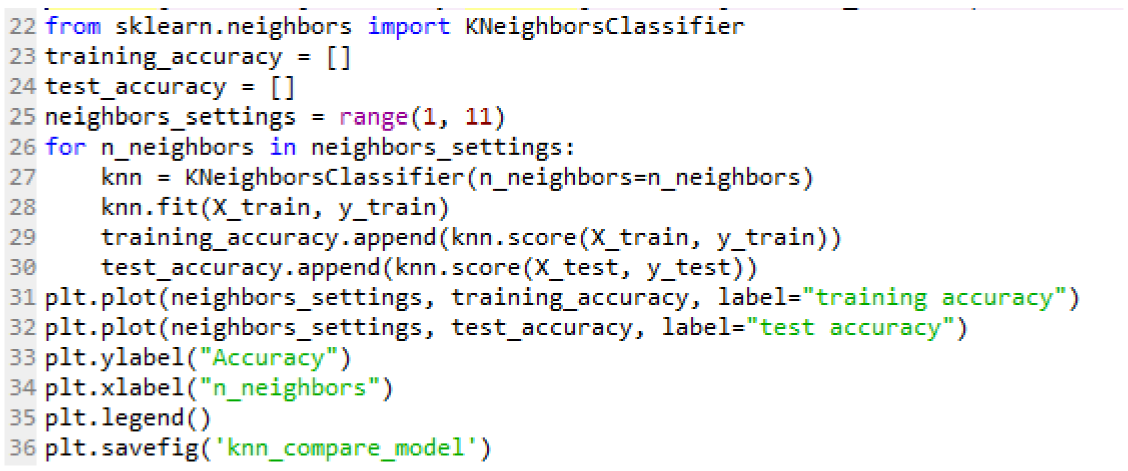


Figure 3.1.1.



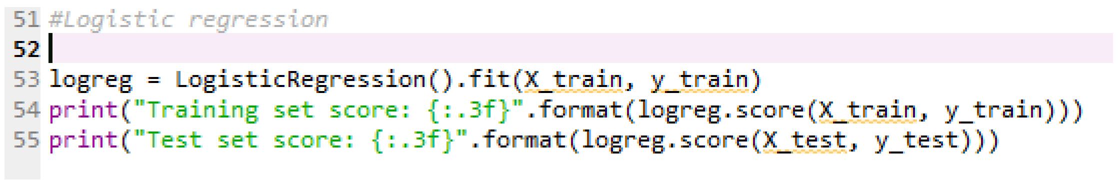
```python
from sklearn.neighbors import KNeighborsClassifier
training_accuracy = []
test_accuracy = []
neighbors_settings = range(1, 11)
for n_neighbors in neighbors_settings:
    knn = KNeighborsClassifier(n_neighbors=n_neighbors)
    knn.fit(X_train, y_train)
    training_accuracy.append(knn.score(X_train, y_train))
    test_accuracy.append(knn.score(X_test, y_test))
plt.plot(neighbors_settings, training_accuracy, label="training accuracy")
plt.plot(neighbors_settings, test_accuracy, label="test accuracy")
plt.ylabel("Accuracy")
plt.xlabel("n_neighbors")
plt.legend()
plt.savefig('knn_compare_model')
```

The above plot shows the training and test set accuracy on the y-axis against the setting of n_neighbors on the x-axis. Considering if we choose one single nearest neighbor, the prediction on the training set is perfect. But when more neighbors are considered, the training accuracy drops, indicating that using the single nearest neighbor leads to a model that is too complex. The best performance is somewhere around 9 neighbors.

The plot suggests that we should choose n_neighbors=9. Here we are:

3.2. Logistic regression

Logistic Regression is one of the most common classification algorithms.



```
#Logistic regression

logreg = LogisticRegression().fit(X_train, y_train)
print("Training set score: {:.3f}".format(logreg.score(X_train, y_train)))
print("Test set score: {:.3f}".format(logreg.score(X_test, y_test)))
```

The default value of C=1 provides with 78% accuracy on the training and 77% accuracy on the test set.

```
Training set score: 0.781
Test set score: 0.771
C:\Users\Ertug\Anaconda3\lib\site-packages\sklearn\linear_mod
0.22. Specify a solver to silence this warning.
  FutureWarning)
```

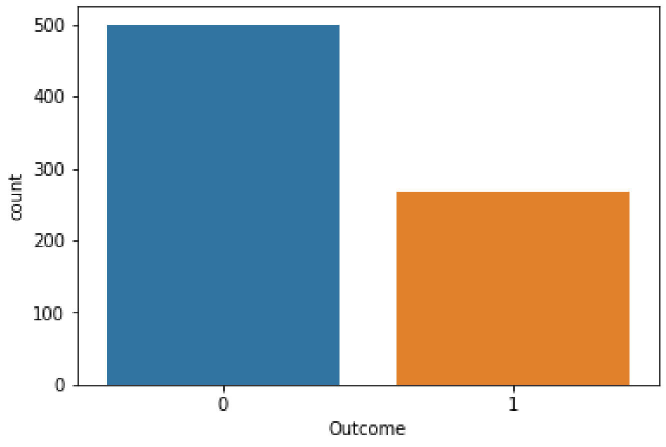
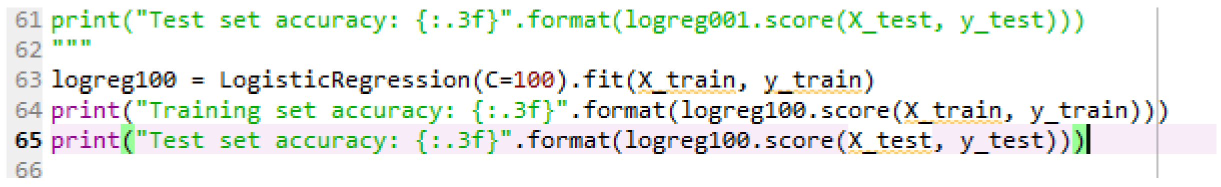
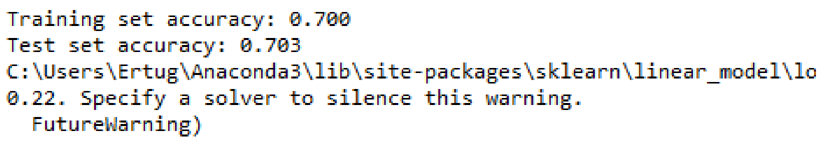


Figure 3.2.1.

Using C=0.01 results in lower accuracy on both the training and the test sets.



```
print("Test set accuracy: {:.3f}".format(logreg001.score(X_test, y_test)))
"""
logreg100 = LogisticRegression(C=100).fit(X_train, y_train)
print("Training set accuracy: {:.3f}".format(logreg100.score(X_train, y_train)))
print("Test set accuracy: {:.3f}".format(logreg100.score(X_test, y_test)))
```



```
Training set accuracy: 0.700
Test set accuracy: 0.703
C:\Users\Ertug\Anaconda3\lib\site-packages\sklearn\linear_model\lo
0.22. Specify a solver to silence this warning.
  FutureWarning)
```

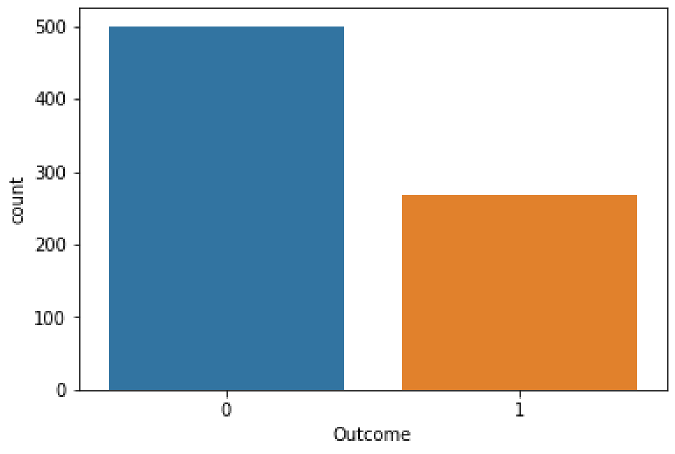


Figure 3.2.2.

Using C=100 results in a little bit higher accuracy on the training set and little bit lower accuracy on the test set, confirming that less regularization and a more complex model may not generalize better than default setting.

Therefore, we should choose default value C=1.

Let's visualize the coefficients learned by the models with the three different settings of the regularization parameter C.

Stronger regularization (C=0.001) pushes coefficients more and more toward zero. Inspecting the plot more closely, we can also see that feature "DiabetesPedigreeFunction", for C=100, C=1 and C=0.001, the coefficient is positive. This indicates that high "DiabetesPedigreeFunction" feature is related to a sample being "diabetes", regardless which model we look at.

```
"""

diabetes_features = [x for i,x in enumerate(diabetes.columns) if i!=8]
plt.figure(figsize=(8,6))
plt.plot(logreg.coef_.T, 'o', label="C=1")
plt.plot(logreg100.coef_.T, '^', label="C=100")
plt.plot(logreg001.coef_.T, 'v', label="C=0.001")
plt.xticks(range(diabetes.shape[1]), diabetes_features, rotation=90)
plt.hlines(0, 0, diabetes.shape[1])
plt.ylim(-5, 5)
plt.xlabel("Feature")
plt.ylabel("Coefficient magnitude")
plt.legend()
plt.savefig('log_coef')
```

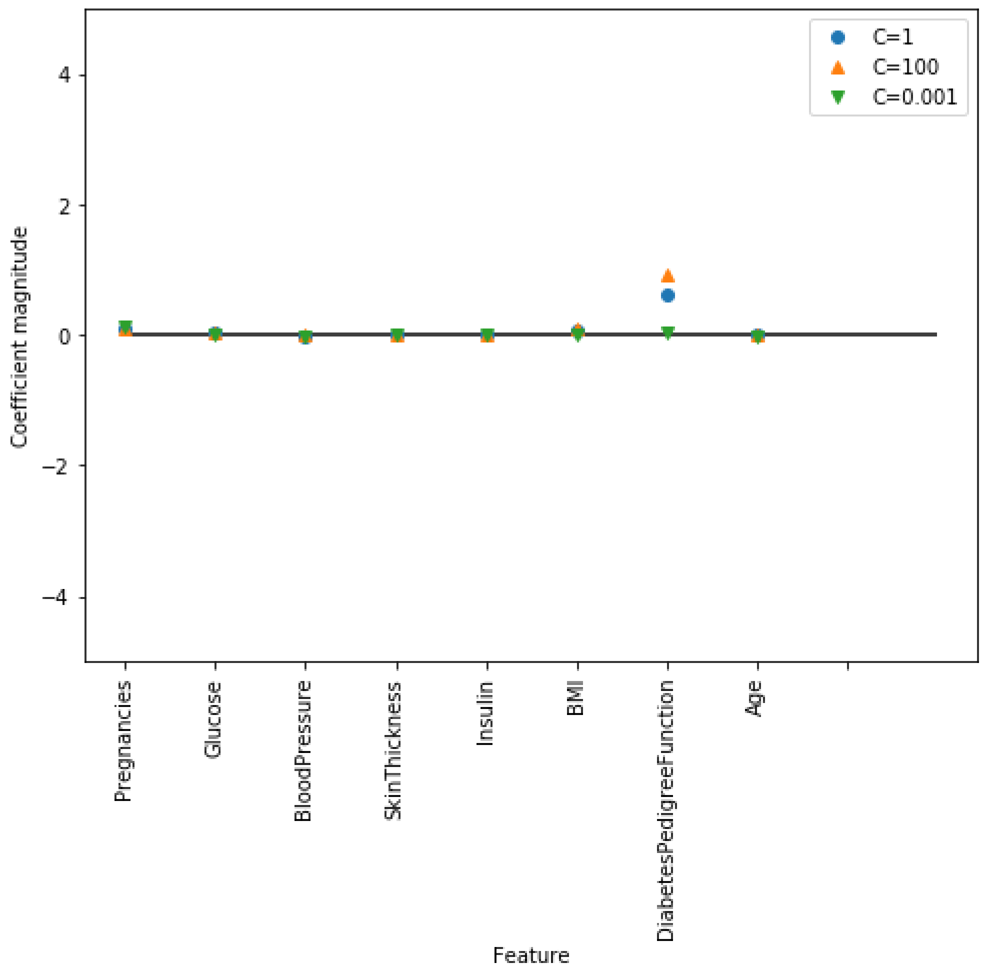
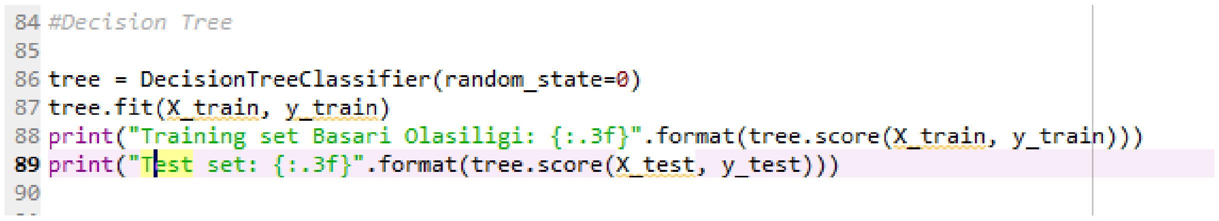


Figure 3.2.3.

3.3. Decision Tree



```
#Decision Tree

tree = DecisionTreeClassifier(random_state=0)
tree.fit(X_train, y_train)
print("Training set Basari Olasiligi: {:.3f}".format(tree.score(X_train, y_train)))
print("Test set: {:.3f}".format(tree.score(X_test, y_test)))

```

The accuracy on the training set is 100%, while the test set accuracy is much worse. This is an indicative that the tree is overfitting and not generalizing well to new data. Therefore, we need to apply pre-pruning to the tree.

We set max_depth=3, limiting the depth of the tree decreases overfitting. This leads to a lower accuracy on the training set, but an improvement on the test set.

```
Training set Basari Olasiligi: 1.000
Test set: 0.714
```

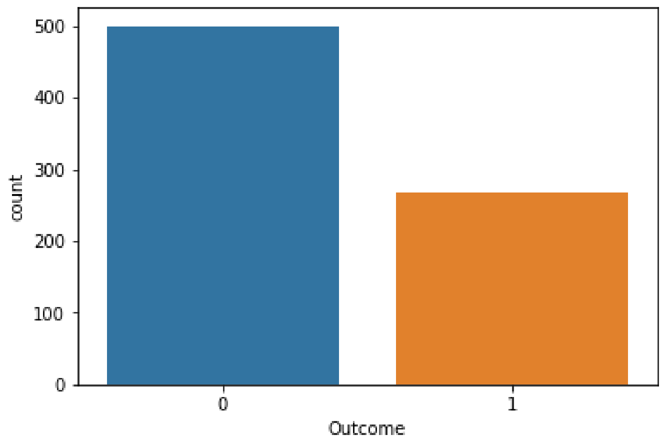
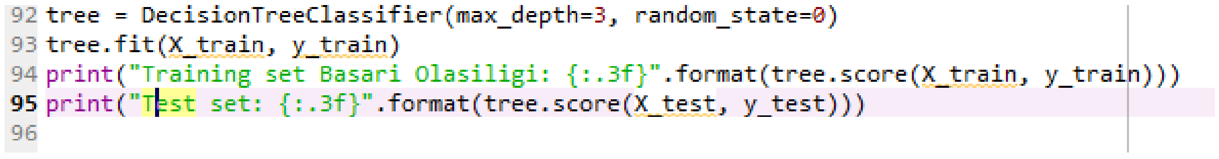


Figure 3.3.1.

3.3.1. Feature Importance in Decision Trees



```
tree = DecisionTreeClassifier(max_depth=3, random_state=0)
tree.fit(X_train, y_train)
print("Training set Basari Olasiligi: {:.3f}".format(tree.score(X_train, y_train)))
print("Test set: {:.3f}".format(tree.score(X_test, y_test)))

```

Feature importance rates how important each feature is for the decision a tree makes. It is a number between 0 and 1 for each feature, where 0 means “not used at all” and 1 means “perfectly predicts the target”. The feature importances always sum to 1:

```
Training set Basari Olasiligi: 0.773
Test set: 0.740
```

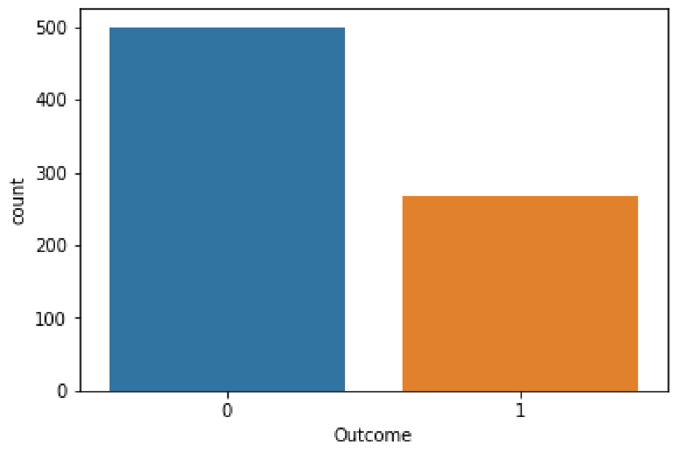
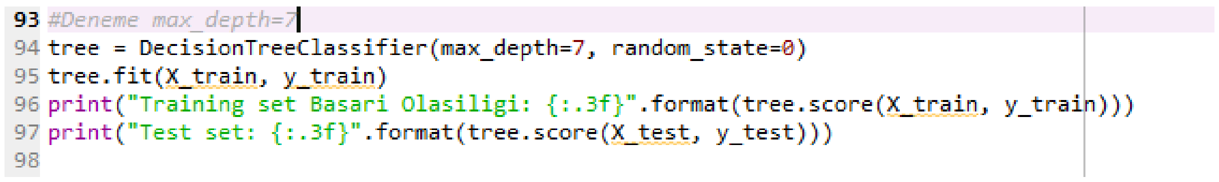


Figure 3.3.1.1.



```python
#Deneme max_depth=7
tree = DecisionTreeClassifier(max_depth=7, random_state=0)
tree.fit(X_train, y_train)
print("Training set Basari Olasiligi: {:.3f}".format(tree.score(X_train, y_train)))
print("Test set: {:.3f}".format(tree.score(X_test, y_test)))
```

```
Training set Basari Olasiligi: 0.908
Test set: 0.740
```

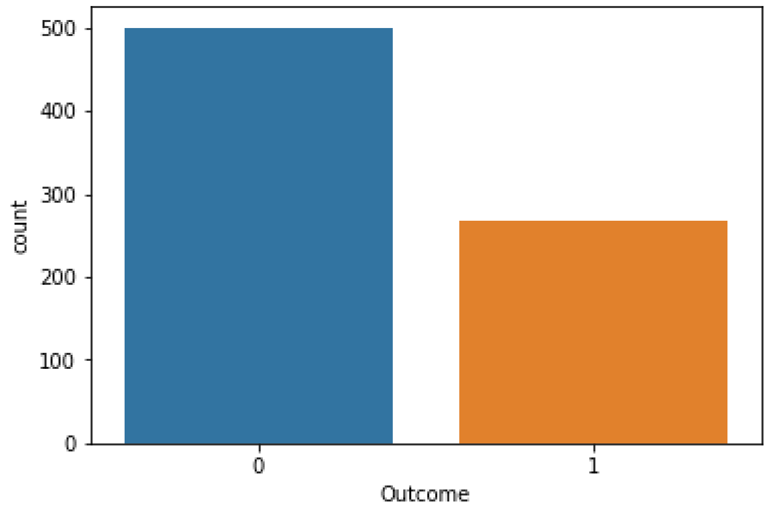
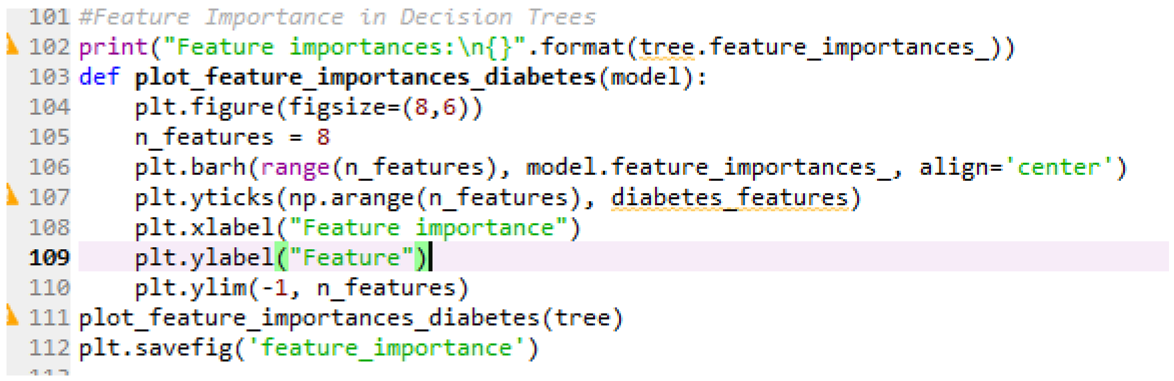


Figure 3.3.1.2.



```
#Feature Importance in Decision Trees
print("Feature importances:\n{}".format(tree.feature_importances_))
def plot_feature_importances_diabetes(model):
    plt.figure(figsize=(8,6))
    n_features = 8
    plt.barh(range(n_features), model.feature_importances_, align='center')
    plt.yticks(np.arange(n_features), diabetes_features)
    plt.xlabel("Feature importance")
    plt.ylabel("Feature")
    plt.ylim(-1, n_features)
plot_feature_importances_diabetes(tree)
plt.savefig('feature_importance')
```

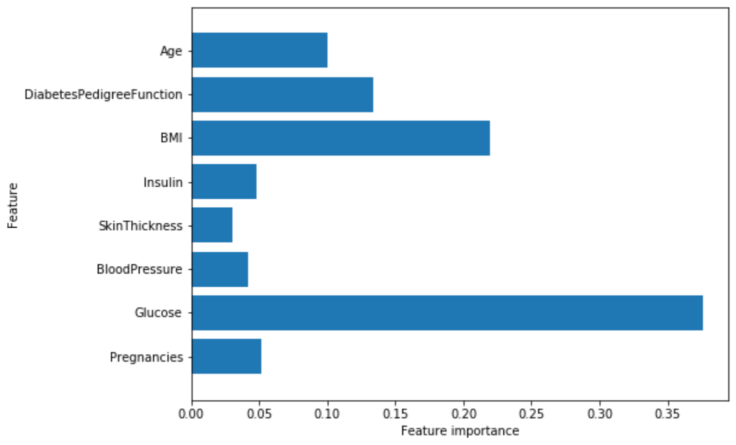


Figure 3.3.1.3.

Feature “Glucose” is by far the most important feature.

```
Accuracy on training set: 1.000
Accuracy on test set: 0.786
```

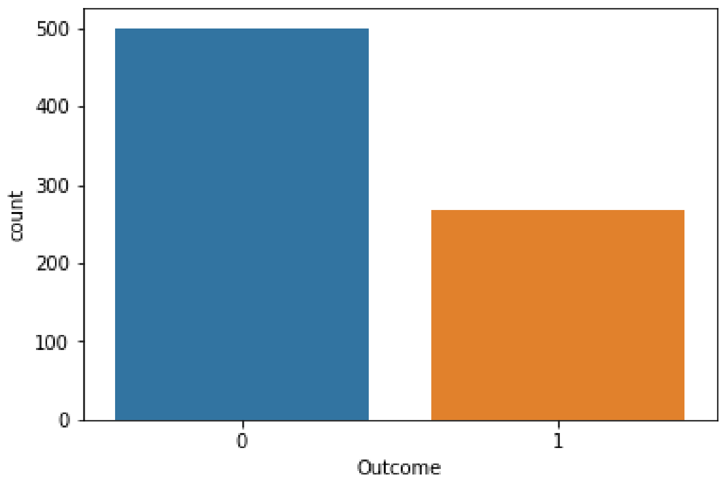


Figure 3.3.1.4.

3.4. Random Forest

Let’s apply a random forest consisting of 100 trees on the diabetes data set:

```
rf1 = RandomForestClassifier(max_depth=3, n_estimators=100, random_state=0)
rf1.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(rf1.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(rf1.score(X_test, y_test)))
```

```
Accuracy on training set: 0.800
Accuracy on test set: 0.755
```

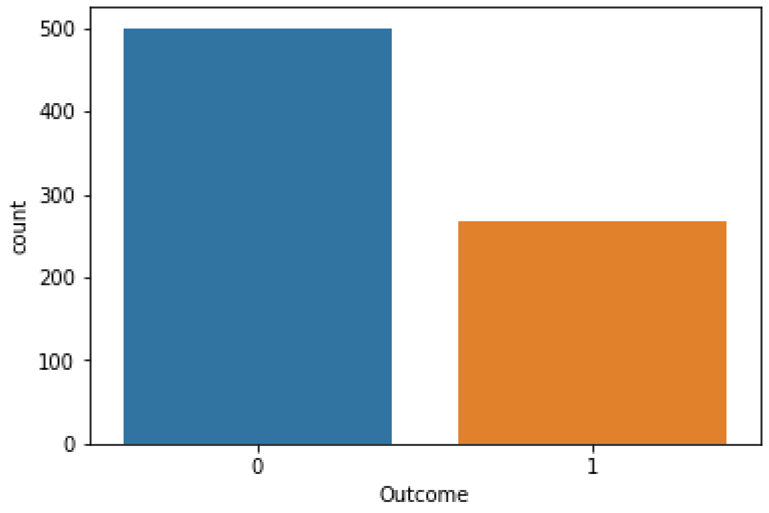


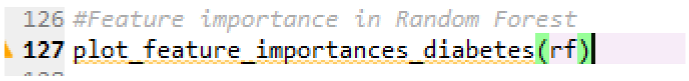
Figure 3.4.1.

The random forest gives us an accuracy of 78.6%, better than the logistic regression model or a single decision tree, without tuning any parameters. However, we can adjust the max_features setting, to see whether the result can be improved.

```
#Random Forest
rf = RandomForestClassifier(n_estimators=100, random_state=0)
rf.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(rf.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(rf.score(X_test, y_test)))
```

It did not, this indicates that the default parameters of the random forest work well.

3.4.1. Feature importance in Random Forest



```
#Feature importance in Random Forest
plot_feature_importances_diabetes(rf)
```

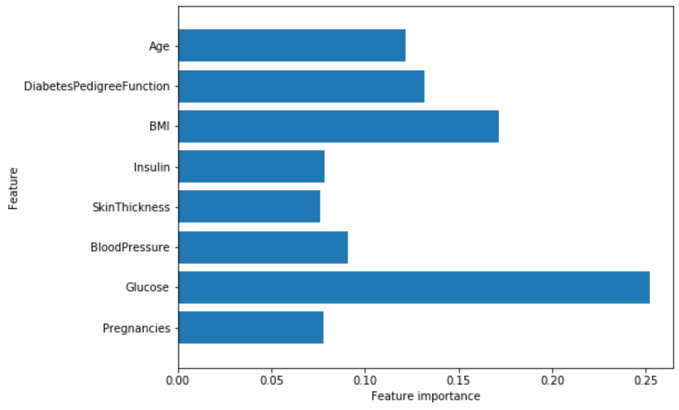


Figure 3.4.1.1.

Similarly to the single decision tree, the random forest also gives a lot of importance to the “Glucose” feature, but it also chooses “BMI” to be the 2nd most informative feature overall. The randomness in building the random forest forces the algorithm to consider many possible explanations, the result being that the random forest captures a much broader picture of the data than a single tree.

```
Accuracy on training set: 0.917
Accuracy on test set: 0.792
```

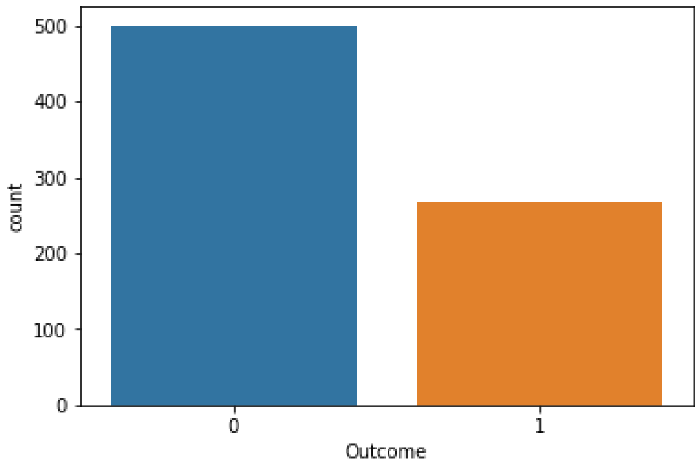


Figure 3.4.1.2.

3.5. Gradient Boosting

```
#Gradient Boosting
from sklearn.ensemble import GradientBoostingClassifier
gb = GradientBoostingClassifier(random_state=0)
gb.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(gb.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gb.score(X_test, y_test)))
```

We are likely to be overfitting. To reduce overfitting, we could either apply stronger pre-pruning by limiting the maximum depth or lower the learning rate:

```
gb1 = GradientBoostingClassifier(random_state=0, max_depth=1)
gb1.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(gb1.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gb1.score(X_test, y_test)))
```

```
Accuracy on training set: 0.804
Accuracy on test set: 0.781
```

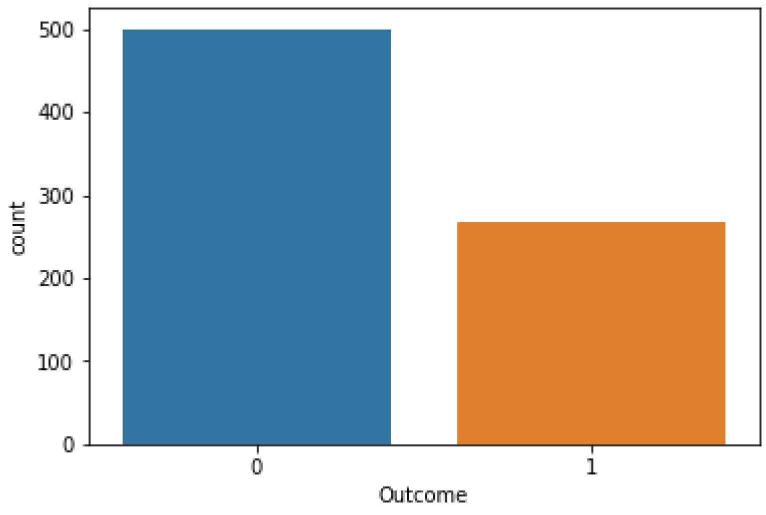


Figure 3.5.1.

```
gb2 = GradientBoostingClassifier(random_state=0, learning_rate=0.01)
gb2.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(gb2.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gb2.score(X_test, y_test)))
```

```
Accuracy on training set: 0.802
Accuracy on test set: 0.776
```

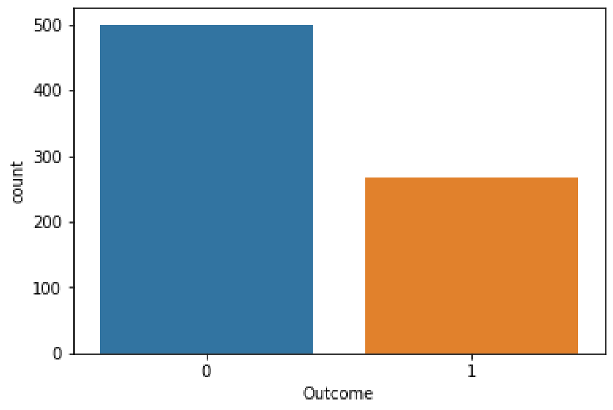


Figure 3.5.2.

Both methods of decreasing the model complexity reduced the training set accuracy, as expected. However, in this case, none of these methods increased the generalization performance of the test set. We can visualize the feature importances to get more insight into our model even though we are not really happy with the model:

```
plot_feature_importances_diabetes(gb1)
```

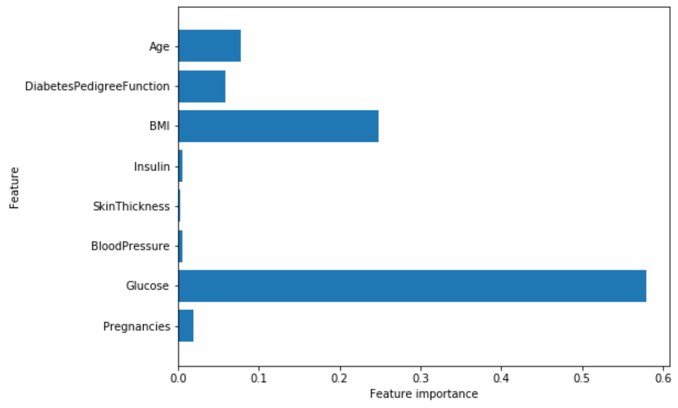
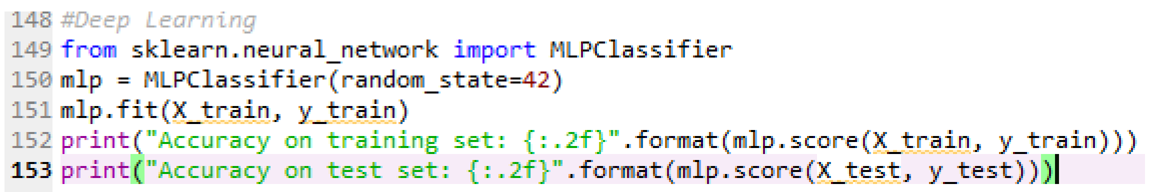


Figure 3.5.3.

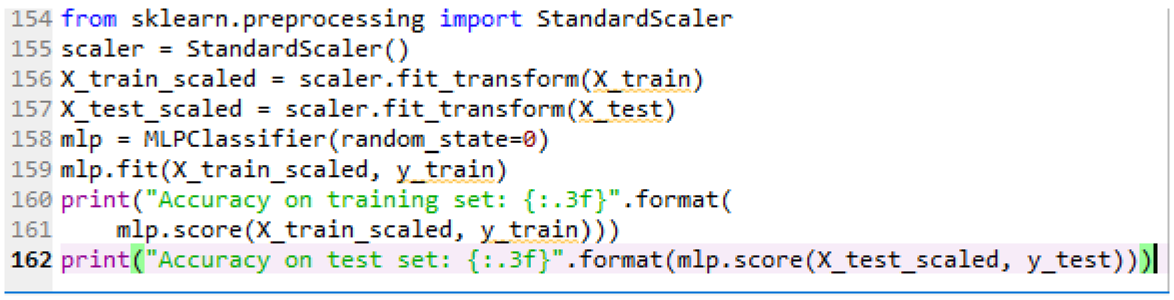
We can see that the feature importances of the gradient boosted trees are somewhat similar to the feature importances of the random forests, it gives weight to all of the features in this case.

3.6. Deep Learning



```
#Deep Learning
from sklearn.neural_network import MLPClassifier
mlp = MLPClassifier(random_state=42)
mlp.fit(X_train, y_train)
print("Accuracy on training set: {:.2f}".format(mlp.score(X_train, y_train)))
print("Accuracy on test set: {:.2f}".format(mlp.score(X_test, y_test)))
```

The accuracy of the Multilayer perceptrons (MLP) is not as good as the other models at all, this is likely due to scaling of the data. deep learning algorithms also expect all input features to vary in a similar way, and ideally to have a mean of 0, and a variance of 1. We must re-scale our data so that it fulfills these requirements.



```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.fit_transform(X_test)
mlp = MLPClassifier(random_state=0)
mlp.fit(X_train_scaled, y_train)
print("Accuracy on training set: {:.3f}".format(
    mlp.score(X_train_scaled, y_train)))
print("Accuracy on test set: {:.3f}".format(mlp.score(X_test_scaled, y_test)))
```

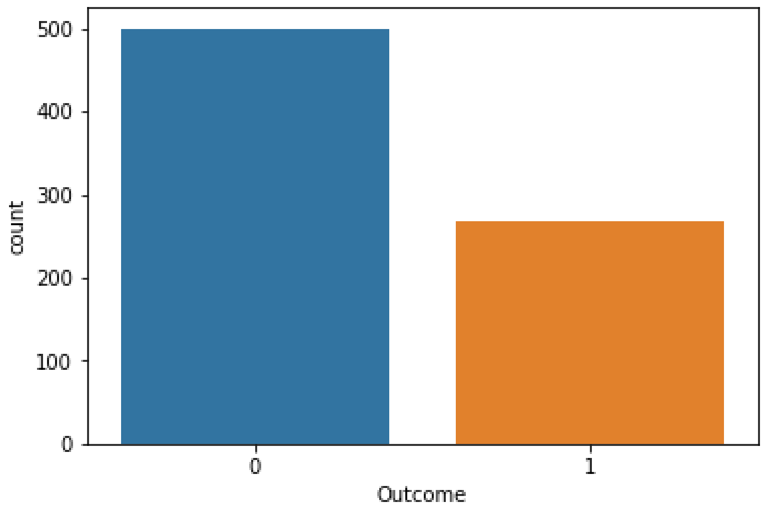
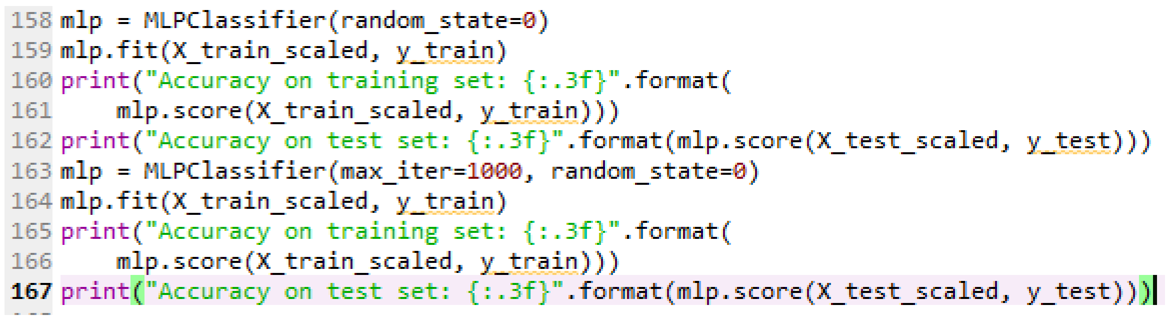
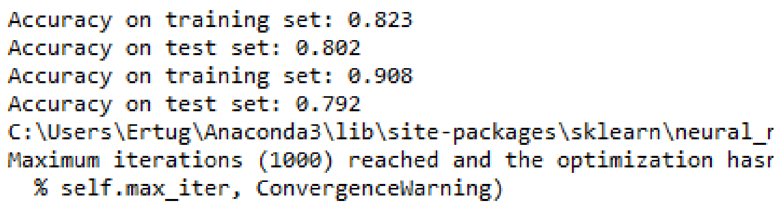


Figure 3.6.1.

Let’s increase the number of iterations:


```
mlp = MLPClassifier(random_state=0)
mlp.fit(X_train_scaled, y_train)
print("Accuracy on training set: {:.3f}".format(
    mlp.score(X_train_scaled, y_train)))
print("Accuracy on test set: {:.3f}".format(mlp.score(X_test_scaled, y_test)))
mlp = MLPClassifier(max_iter=1000, random_state=0)
mlp.fit(X_train_scaled, y_train)
print("Accuracy on training set: {:.3f}".format(
    mlp.score(X_train_scaled, y_train)))
print("Accuracy on test set: {:.3f}".format(mlp.score(X_test_scaled, y_test)))
```


```
Accuracy on training set: 0.823
Accuracy on test set: 0.802
Accuracy on training set: 0.908
Accuracy on test set: 0.792
C:\Users\Ertug\Anaconda3\lib\site-packages\sklearn\neural_
Maximum iterations (1000) reached and the optimization has
  % self.max_iter, ConvergenceWarning)
```

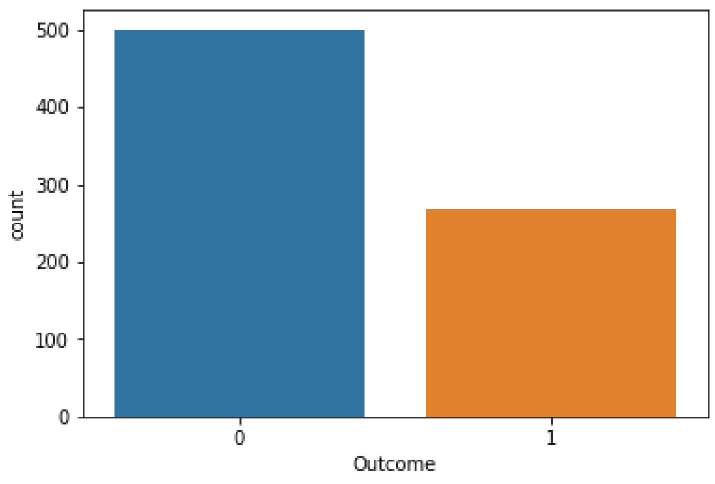
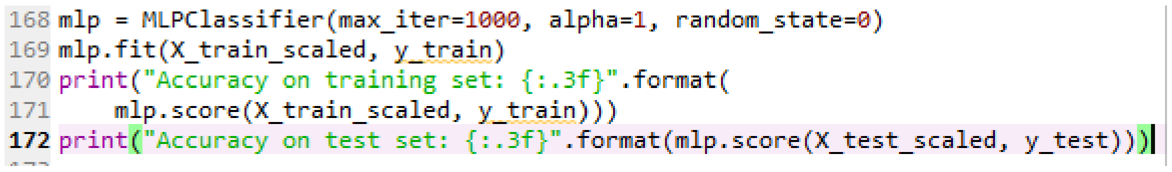


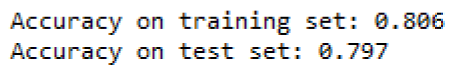
Figure 3.6.2.

Increasing the number of iterations only increased the training set performance, not the test set performance.

Let’s increase the alpha parameter and add stronger regularization of the weights;



```python
mlp = MLPClassifier(max_iter=1000, alpha=1, random_state=0)
mlp.fit(X_train_scaled, y_train)
print("Accuracy on training set: {:.3f}".format(
    mlp.score(X_train_scaled, y_train)))
print("Accuracy on test set: {:.3f}".format(mlp.score(X_test_scaled, y_test)))
```



```
Accuracy on training set: 0.806
Accuracy on test set: 0.797
```

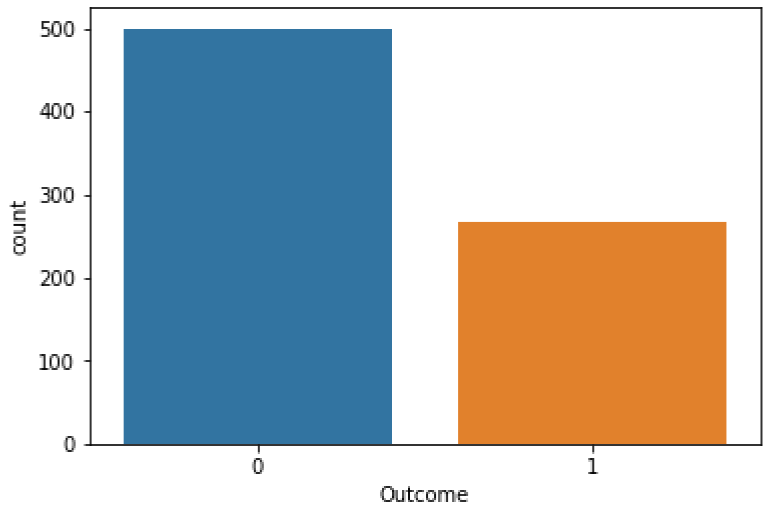
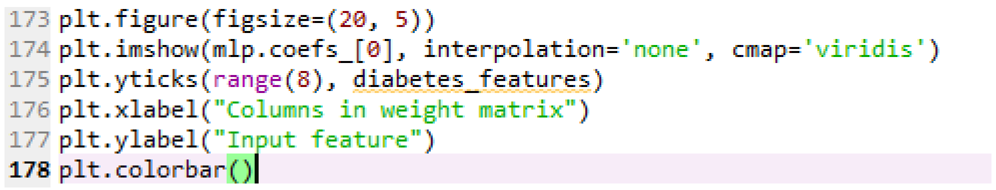


Figure 3.6.3.

The result is good, but we are not able to increase the test accuracy further.

Therefore, our best model so far is default deep learning model after scaling.

Finally, we plot a heat map of the first layer weights in a neural network learned on the diabetes data set.



```
plt.figure(figsize=(20, 5))
plt.imshow(mlp.coefs_[0], interpolation='none', cmap='viridis')
plt.yticks(range(8), diabetes_features)
plt.xlabel("Columns in weight matrix")
plt.ylabel("Input feature")
plt.colorbar()
```

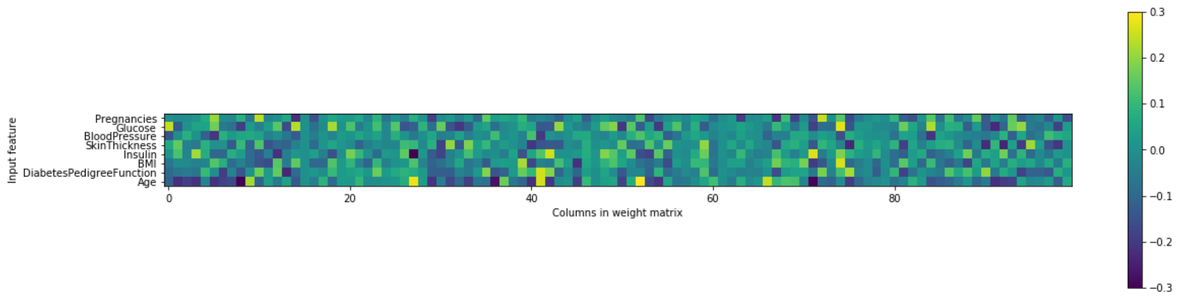


Figure 3.6.4.

From the heat map, it is not easy to point out quickly that which feature have relatively low weights compared to the other features.

4. Summary

There are a wide range of machine learning models, advantages and disadvantages for classification and regression, and we've tried how to control the model complexity for each. In most algorithms, we found that adjusting the correct parameters is important for good performance.

We need to know how to implement, adjust, and analyze the models we have implemented above.