George Mason University

Learning From Data Fall 2024

Computer Exercise #5

Assigned: September 27, 2024
Due Date: October 07, 2024

Overview: The main focus of this computer exercise is on logistic regression, and how it may be used for diabetes prediction. Most of the code needed for these exercises is available in a Jupyter notebook that is distributed with this assignment. Your lab report should also be submitted in a Jupyter notebook with full documentation of what you have done (and why) along with answers to or discussion about the questions that are given. The due date for this is Monday, October 07 at midnight. As will all previous assignments, no late submissions will be accepted for any reason. Submit early and submit often to at least have a partial submission in on time.

The purpose of this lab is to make you go beyond the theory of logistic regression developed in class, and to see how it works in practice. This is your opportunity to experiment, and along the way (if you put in the effort) you may find yourself challenged to understand logistic regression on a deeper level. You will also be learning about some performance metrics in addition to the empirical risk.

Comment: As in the previous computer exercise, you will be given some basic code to minimize the chances of typographical errors, or instructions on how to perform some task. It will be up to you to piece things together to perform the experiments that you are asked to do.

Note: You should view the tasks that you are given and the questions that you are asked as guidelines for what to do. Feel free to experiment with other ideas that you might have, and you are encouraged to do so. View this as an open-ended project, and avoid using this assignment as one that is scripted in terms of what to do. Your grade will depend, in part, on how much creativity you show.

In this computer exercise, you will be using logistic regression to train a predictor of whether or not a patient has diabetes using the Pima Indian dataset, and to predict the risk factors associated with diabetes. Each data sample includes information about various medical attributes such as the number of pregnancies, plasma glucose concentration, tricep skinfold thickness, body mass index (BMI), diastolic blood pressure, 2-h serum insulin (serum-insulin), age and diabetes pedigree function. Each data sample has a classification label of 1 if an individual was diagnosed with type 2 diabetes and 0 otherwise. There are 268 (34.9%) diabetes patients in the data set. Five covariates, insulin, glucose, BMI, skin thickness, and blood pressure, contain at least one missing value (indexed by zero), which is not meaningful, and so these zero values were replaced with the corresponding median values.

Diabetes is one of the most common human diseases and has become a significant public health concern worldwide. There were approximately 450 million people diagnosed with diabetes that resulted in around 1.37 million deaths globally in 2017. More than 100 million US adults live with diabetes, and diabetes was the seventh leading cause of death in the US in 2020. One in ten US adults have diabetes, and if the current trend continues, it is projected that as many as one in three US adults could have diabetes by 2050. An individual at high risk of diabetes may not be aware of the risk factors, and given the high prevalence and severity of diabetes, researchers are interested in finding the most common risk factors of diabetes.

Computer Exercise 5.1 (Predicting Diabetes Using Logistic Regression):

In this computer exercise you will be using **logistic regression** to predict whether or not a patient has diabetes. Logistic regression approaches the **binary** classification of data directly by finding the parameters of a model for the probability that a data sample has a target value of $Y = 1$,

where the model has the form

$$\Pr\{Y = 1|\boldsymbol{x}\} = \frac{1}{1 + \exp\{-\boldsymbol{w}^T\boldsymbol{x}\}}$$

where $\boldsymbol{w} = [w_0, w_1, \ldots, w_p]^T$ with feature vectors $\boldsymbol{x} = [1, x_1, x_2, \ldots, x_p]^T$. The extension to multiple classes is done using softmax regression where the output of softmax regression is a vector of probabilities, $\Pr\{Y = k|\boldsymbol{x}\}$, one for each class. Classification is then done by selecting the class that has the largest probability.

1. Getting Started

For this exercise, in addition to the basic imports that have used before, here we will be introducing two new ones. The first is `pandas`, which is a package that allows us to work with **data frames**. Data frames in Python come with the Pandas library, and they are defined as two-dimensional labeled data structures with columns of potentially different types. The second new import is `seaborn`, which is a Python data visualization library based on `matplotlib` that provides a high-level interface for drawing attractive and informative statistical graphics.

So the starting point is to import the following packages, including the `LogisticRegression` class from the `sklearn.linear_model` library:

```
#Common imports
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
```

2. The Dataset

As mentioned in the introduction, the diabetes dataset that we will be using is the Pima Indian dataset, which has been used extensively in machine learning projects to develop a method for predicting diabetes based on a number of features. It was used as recently as 2021 in a paper by Joshi et. al. in 2021,[1] and some of the experiments in this exercise are drawn from or motivated by this paper.

The datset is stored in a .csv file called `diabetes2.csv` that is distributed with this assignment, and it may be imported for use with scikit-*learn* using pandas,

```
diabetesDF = pd.read_csv('diabetes.csv')
```

Note that `diabetesDF` is a DataFrame, and not a `numpy` array. An advantage of Pandas data frames is that they can handle mixed data types, with some columns contain text and others containing numbers. The columns with numbers may be converted easily to a `numpy` array using the function `numpy.asarray` or to a matrix with `numpy.asmatrix`. Using the `shape` function you will see that the dataset is (769,9).

Having loaded the diabetes dataset, the `head` function may be used to show the first five records of the dataset in order to see exactly what the data frame looks like

```
print(diabetesDF.head())
```

As shown in Fig. 1, each record has eight features along with the outcome. These features are:

[1] Joshi, R.D.; Dhakal, C.K. Predicting Type 2 Diabetes Using Logistic Regression and Machine Learning Approaches. Int. J. Environ. Res. Public Health 2021, 18, 7346. https://doi.org/10.3390/ ijerph18147346

diabetes2

Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
6	148	72	35	0	34	0.627	50	1
1	85	66	29	0	27	0.351	31	0
8	183	64	0	0	23	0.672	32	1
1	89	66	23	94	28	0.167	21	0
0	137	40	35	168	43	2.288	33	1

Figure 1: The diabetes dataset.

1. **Pregnancies**: Number of times pregnant
2. **Glucose**: Plasma glucose concentration over 2 hours in an oral glucose tolerance test
3. **BloodPressure**: Diastolic blood pressure (mm Hg)
4. **SkinThickness**: Triceps skin fold thickness (mm)
5. **Insulin**: 2-Hour serum insulin (mu U/ml)
6. **BMI**: Body mass index (weight in kg/(height in m)2)
7. **DiabetesPedigreeFunction**: Diabetes pedigree function (a function which scores likelihood of diabetes based on family history)
8. **Age**: Age (years)

The final column, **Outcome**, is the class variable (0 if non-diabetic, 1 if diabetic).

3. Data Preparation

Before using a dataset to learn a hypothesis, there are typically some data preparation steps that are necessary and important. First, it is advisable to make sure that there is no missing data, such as missing features in data samples since missing features may degrade the performance of the classifier. If there are missing features, one way to deal with this is to to replace missing values with the mean value for that feature.

To check to see if the data has any missing entries (null values), use the `info` method,

```
diabetesDF.info()
```

and what we find is that there is no missing data,

```
# output shown below
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
Pregnancies                 768 non-null int64
Glucose                     768 non-null int64
BloodPressure               768 non-null int64
SkinThickness               768 non-null int64
Insulin                     768 non-null int64
BMI                         768 non-null float64
DiabetesPedigreeFunction    768 non-null float64
Age                         768 non-null int64
Outcome                     768 non-null int64
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

However, some of the records have missing values that have been set to zero, (In Fig. 1, for example, we see that some of the records have `Insulin = 0`). For now, we will not worry about these missing entries, although you classifier may perform better if these zero values are modified.

The next step is to split the dataset into a **training set** and a **test set**. If the number of experiments that are to be run is large, or if it is necessary to search over a set of different hyperparameters to find the best one, then the dataset should be divided into three sets:[2] a training set, a **validation set**, and a test set. Since we are working with DataFrames, and since there will be no need for a validation set, `diabetesDF` can be split into a training set and a test set as follows:

```
DF_train = diabetesDF[:N1]
DF_test = diabetesDF[N1:]
```

where `N1` defines the desired size of the training set with the remaining samples being put into the test set (the diabetes dataset has a total of 767 records).

Next, the training and test sets are separated into two DataFrames, one containing the features and other target values, and these are then converted into NumPy arrays, which is the data format expected from the Scikit-*Learn* functions,

```
X_train = np.asarray(DF_train.drop('Outcome',1))
y_train = np.asarray(DF_train['Outcome'])
X_test = np.asarray(DF_test.drop('Outcome',1))
y_test = np.asarray(DF_test['Outcome'])
```

The final step in data preparation is data scaling and normalization. With few exceptions, machine learning algorithms do not perform well when the numerical attributes of the features have different scales. Therefore, **feature scaling** is an important step in learning a classifier. Scaling the data also makes it easier to understand the importance of each feature when looking at the model weights.

There are two common methods for scaling feature vectors: **min-max scaling** and **standardization**. Min-max acaling is simple and involves shifting and scaling the data set so each feature has a range of values between zero and one. Scikit-*Learn* provides a transformer called `MinMaxScaler` to do this.

Standardization is different in that it scales each feature so that it has zero mean and unit variance. This is done by first subtracting the mean from each feature, and then scaling each feature by its variance. Scikit-*Learn* provides a transformer for standardization called `StandardScaler`. Unlike the min-max scaler, standardization does not produce features that are bounded between zero and one, which may be a problem for some algorithms such as neural networks that often expect an input in the range from zero to one. On the other hand, standardization is not affected as much by outliers as the `MinMaxScaler`.

An example of how to scale data that is stored in the array `X` using the min-max scaler is:

```
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
scaler=MinMaxScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)
```

To use the standardization scaler, define

```
scaler=StandardScaler()
```

[2]This is will be discussed in more detail in our discussions of cross-validation.

It is important to note that, in both cases, once `scaler.fit(X)` is called, this sets the scaling and any other calls to the transformer will use the same scaling. To use different scalings on different data sets, a separate call to `scaler.fit` must be done.

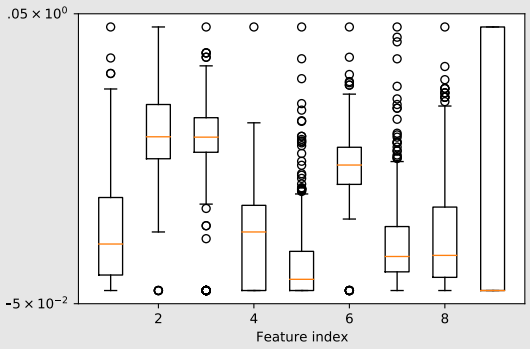
Data Preparation

(a) Separate your DataFrames into a training set of size $N_1 = 600$, with a test set containing all of the remaining samples, and convert them into NumPy arrays to form the labeled training set `(X_train, y_train)` and labeled test set `(X_test, y_test)`.

(b) Scale the training set using one of the methods above. Check to see what the scaled features look like by making box plots of the features before and after scaling. This may be done for the training set `X_train` as follows:

```
plt.boxplot(X_train, manage_ticks=False)
plt.yscale("symlog")
plt.xlabel("Feature index")
plt.ylabel("Feature magnitude")
plt.show()
```

An example of what your box plot might look like is shown below. You may wish to read the documentation on `boxplot` to see how to interpret this plot.



Do you see any issues or concerns with the way the data is scaled for the two types of scalers?

(c) Next, scale the test set. Here you have two options to consider.

i. Scale the training set and then scale the test set in exactly the same way, or

ii. Scale the training set and the test set independently so that they both have features in the range zero to one in the case of the MinMax scaler, or have zero mean and unit variance for the standardization scaler.

Decide which approach you should use, explain why you made this choice, and then scale the test set.

When you are satisfied with the scaling that was done, proceed to the next part.

4. Exploring the Data

It is useful to explore the dataset to get a feel of what the data looks like, and to see what relationships might exist among the features.[3] We have seen an example where it is possible that not all features are relevant or useful for classification (e.g. the corner pixels in the digits dataset for digit recognition), and that dimension reduction, if feasible, is often a good idea.

[3]It is important to not make any decisions about what kind of classifiers should be used by looking at the data. This is called **data snooping**.

The diabetes dataset has eight features, and it is not clear whether or not all of them are useful or needed for effective classification.[4] Some features may be correlated with others while others may not provide any useful information for distinguishing one class from another.

Correlation is a measure of the linear dependence between two random variables, and if the correlation between two features is high (a value that is close to ± 1), then there is an approximate linear relationship between the two variables, and knowledge of one makes the information provided by the second not as useful as it would be if the correlation was zero.[5]

Experiment

(a) Explore the correlation between features (and the target values) using the `corr` method of the diabetes DataFrame,

```
corr = diabetesDF.corr()
```

The output will be an 8×8 matrix of correlations similar to what is shown in the table below for three features.

	Pregnancies	Glucose	Blood Pressure
Pregnancies	1.000000	0.129459	0.141282
Glucose	0.129459	1.000000	0.152590
Blood Pressure	0.141282	0.152590	1.000000

The entries along the main diagonal are equal to one because each feature is perfectly correlated with itself. As shown here, the correlation between `Glucose` and `Blood Pressure` is 0.15259.

In looking at all eight features, what pairs of features have the largest correlation?

What are the features that are most correlated with the target outputs? These will most likely be the best candidates for detecting diabetes, assuming that they are not highly correlated with each other.

(b) As an aid in visualization of the correlations, create a heat map as follows:

```
sns.heatmap(corr, xticklabels=corr.columns, yticklabels=corr.columns)
```

Those cells that are the brightest red are the ones that have the largest correlation, whereas those that are black have close to zero correlation.

Although it is may be difficult to visualize all eight features and how they interact with each other, there are other visualization tools that can be helpful. One of these is `pairplot`, which shows pairwise relationships between features along with the marginal distributions of each feature. An example is shown in Fig. 2 for three of the diabetes features that was created as follows:

```
sns.pairplot(DF_train, vars=['BMI', 'BloodPressure', 'Age'], hue='Outcome')
plt.show()
```

Note that the plots along the diagonal are the histograms of each feature for each class, and the plots off the main diagonal are scatter plots for two features at a time. A pair-plot may be produced for all of the features by removing the list given in `vars`, and this will produce an 8×8 array of plots.

[4] Although eight is not a large number, it may not be a bad idea to keep all of them, given the size of the data set, but keep in mind the curse of dimensionality

[5] Correlations above 0.5 might be considered high in some cases.

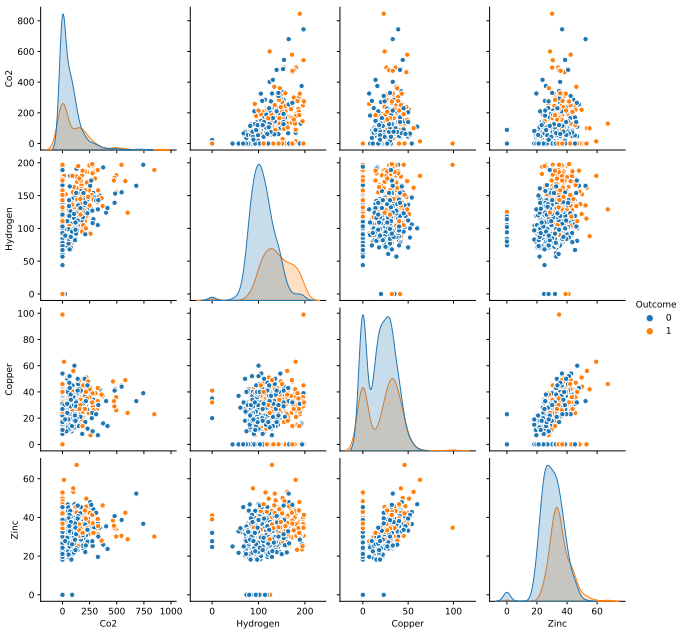


Figure 2: Pair plot for three of the features in the diabetes dataset.

Experiment

Using the training set `(X_train,y_train)`, create some pair plots to get a better feeling of the relationships (if any) between the features. Are you able to learn anything about the data that might be useful? Although this is not something that you would generally do in practice, from your pair plots you might be able to get a feeling about which features would be the best for learning a classifier.

Another tool that might be useful in understanding the data is `displot` in the `seaborn` library that produces kernel density estimates of the feature data. To create a plot of the KDE estimate of the `Glucose` feature shown in Fig. 3 you would type

```
sns.displot(DF_train['Glucose'].dropna(),kde=True)
```

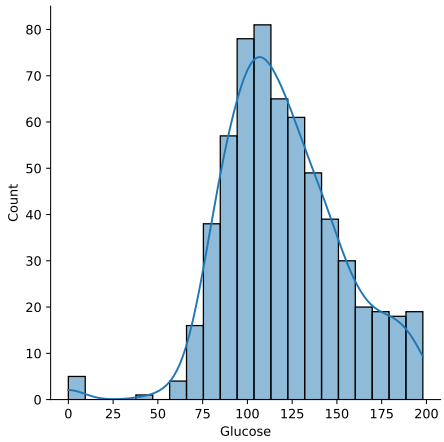
Similarly, a two-dimensional KDE estimate can be generated as shown in Fig. 3(b) for `Glucose` and `Age` as follows,

```
sns.displot(data=DF_train, x='Glucose',y='Age',kind='kde')
```

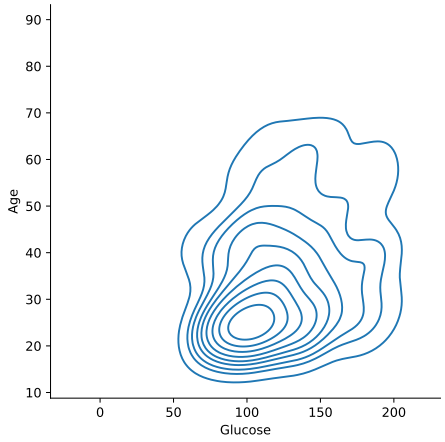
This does not tell us much, and seems to indicate that using these two features will not produce an effective classifier. However, note that all of the data is merged together in the computation of the KDE. What is needed is to find the 2-D KDE for each class separately, and compare them. This can be done by setting `hue="Outcome"`. This will produce two KDE estimates, one for each outcome as illustrated in Fig. 3(c). From this plot it looks like it may be possible to use a linear classifier using `Glucose` and `BMI` as the features. What the accuracy is of such a classifier would need to be evaluated.

5. Logistic Regression

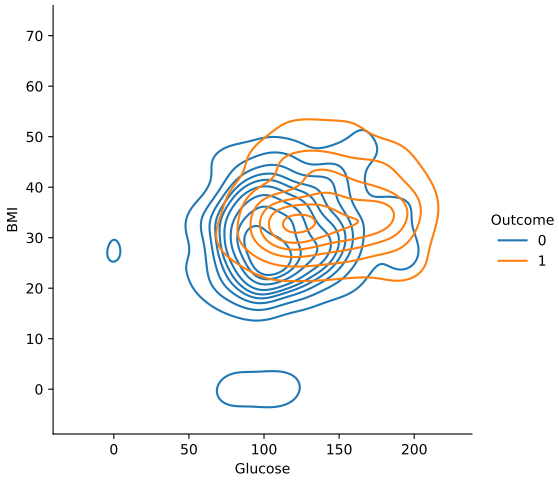
With a scaled training set `(X_train, y_train)`, we are ready to use logistic regression to learn a classifier, which may be done as follows:



(a) `Glucose`



(b) `Glucose` and `Age`



(c) `Glucose` and `Age` separated by class.

Figure 3: Kernel density estimates of diabetes features.

```
# Here, all parameters not specified are set to their defaults
# You may want to define different values for these parameters
clf = LogisticRegression()
clf.fit(X_train, y_train)
```

Having a model for $\Pr\{Y = 1|\boldsymbol{x}\}$, using the `pred` method a data sample $\boldsymbol{x}$ may be classified as having a target value of $Y = 1$, which corresponds to the case $\Pr\{Y = 1|\boldsymbol{x}) > 0.5$, or a target value of $Y = 0$ if $\Pr\{Y = 1|\boldsymbol{x}) < 0.5$. Classification of all samples in the test set may be done as follows:

```
y_test_pred = clf.predict(X_test)
```

Instead of performing classification, it is also possible to find the probability that each sample in the test set has a target value of $Y = 1$ as follows[6]

```
y_test_prob = clf.pred_proba(X_test)
```

[6]Note that for each sample, a vector of length two is output, with the first being the probability that $Y = 1$ and the second that $Y = 0$. Clearly, the sum of these two will be equal to one.

In addition, the mean accuracy on the test set may be found using the `score` method:

```
accuracy = clf.score(X_test, y_test)
print("accuracy = ", accuracy * 100, "%")
```

Experiment

(a) Use logistic regression on the training set to find a model for $\Pr\{Y = 1\}$ where $Y = 1$ is the class associated with the outcome `Has Diabetes`.

(b) What is the accuracy of the classifier on the test set?

(c) Wow many samples in the test set have $\Pr\{Y = 1\} \approx 0.5$, i.e., for how many samples is the confidence in the classification not very high?

(d) Are there any conclusions that you can make about the effectiveness of your classifier for predicting diabetes?

6. Performance: Accuracy, Confusion Matrix, Precision and Recall

Since classification accuracy provides no information about what types of errors are being made, another and often better way to evaluate classifier performance is with a **confusion matrix**. The confusion matrix counts the number of times that instances from one class is classified as another class. Each row represents an actual class and each column represents a predicted class. For a confusion matrix C, the element $c_{i,j}$ is the number of samples in the data set that are in class i and are predicted to be in class j. Thus for a binary classifier, the confusion matrix has two rows and two columns. The number of objects belonging to class 1 that are classified correctly is $c_{1,1}$ and the number that are classified incorrectly is $c_{1,2}$. Similarly, the number of objects belonging to class 2 that are classified correctly is $c_{2,2}$ and the number that are classified incorrectly is $c_{2,1}$.

To compute the confusion matrix, one first needs a set of predictions. Using the predictions `y_test_pred` on the test set generated in the previous part, the confusion matrix class may be used to generate a confusion matrix:

```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test, y_test_pred)
```

Printing the output for this example will produce a 2×2 array that might look like this:

```
array([[ 9013,  95],
       [ 296,  596])
```

Experiment

(a) Find the confusion matrix for the logistic regession classifier that you have designed and discuss your results.

(b) Does the sum of the numbers in each row have any meaning? What about the sum of the numbers in each column? What should the sum of all the numbers in C be equal to?

(c) Are your confusion matrices close to being symmetric? Should they be?

The confusion matrix provides a lot of information, in general, especially when there is a large number of different classes. Sometimes it is preferable to have a more concise metric. One that is often used is the accuracy of the positive predictions. This is called the **precision** of the classifier, and is defined as

$$\text{Precision} = \frac{TP}{TP + FP}$$

where TP is the number of true positives and FP is the number of false positives. Here, a true positive would be correctly classifying a patient as having diabetes, and a false positive would be incorrectly classifying a patient as not having diabetes. A trivial way to have perfect precision is to make one single positive prediction and ensure that it is correct. This would not be very useful since the classifier would ignore all but one positive instance. Therefore, precision is typically used along with another metric called **recall**, which is the fraction of the number of positive instances that are correctly detected by the classifier,

$$\text{Recall} = \frac{TP}{TP + FN}$$

In other words, *recall* is how many of the true positives are recalled (found). Scikit-*Learn* provides several functions to compute classifier metrics, including precision and recall:

```
from sklearn.metrics import precision_score, recall_score
precision_score(y_test, y_test_pred)
recall_score(y_test, y_test_pred)
```

It is sometimes convenient to combine precision and recall into a single metric called the F_1 score. This makes it easier to compare two classifiers. The F_1 score is the harmonic mean of precision and recall,

$$\frac{1}{F_1} = \frac{1}{\text{precision}} + \frac{1}{\text{recall}}$$

The mean treats all values equally, while the harmonic mean gives more weight to low values. A classifier will get a high F_1 score only if both recall and precision are high.

From the precision and recall scores, it is easy to compute the F_1 score, or scikit-*learn* may be used:

```
from sklearn.metrics import f1_score
f1_score(y_test, y_test_pred)
```

Assignment:

(a) Find the precision and recall for your classifier, and compute the F_1 score.

(b) Comment on your results. In view of these metrics, how good is your classifier?

Another useful tool in scikit-*learn* is the **classification report** that generates a table of precision, recall, F_1 score, and accuracy along with macro and weighted averages of precision and recall. This report may be generated as follows:

```
predictions = clf.predict(X_test)
from sklearn.metrics import classification_report
print(classification_report(y_test,predictions))
```

For example, with three-class classification problem with

```
y_true = [0, 1, 2, 2, 2]
y_pred = [0, 0, 2, 2, 1]
```

the classification report would be:

	precision	recall	f1-score	support
class 0	0.50	1.00	0.67	1
class 1	0.00	0.00	0.00	1
class 2	1.00	0.67	0.80	3
accuracy			0.60	5
macro avg	0.50	0.56	0.49	5
weighted avg	0.70	0.60	0.61	5

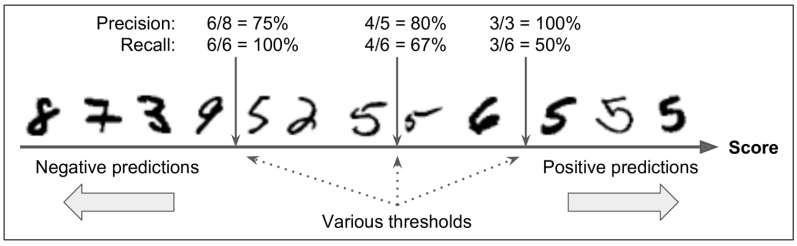
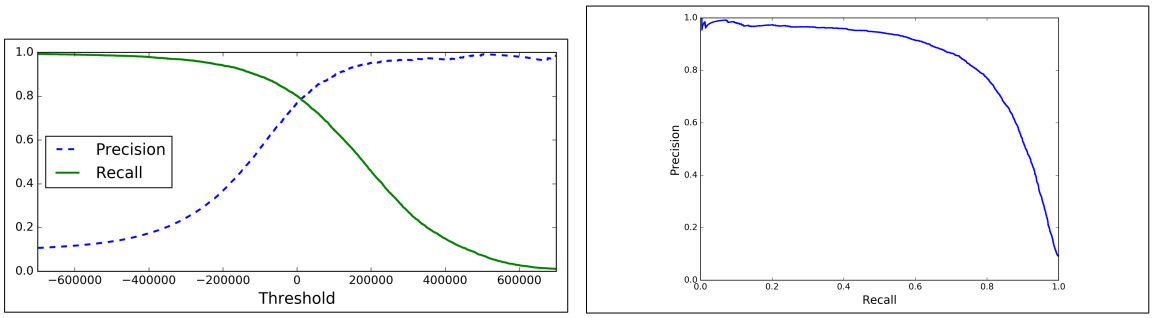


Figure 4: Decision and recall versus threshold for the `Five` versus `Not Five` classifier.



(a) Precision and Recall versus a threshold.

(b) Precision versus recall for various thresholds.

Figure 5: Dependence of precision and recall on the classifier threshold.

Read the documentation on how to read this table.

Another measure of performance for a binary classifier is the ROC curve.[7] An ROC curve is a plot of the **true positive rate** of the classifier versus the **false positive rate** as a function of the classifier threshold. A short introduction to the ROC is given below.

In the binary classification of digits using a `five` versus `not five` classifier, the classifier makes a decision based on the rule,

$$g(\boldsymbol{x}) \underset{\texttt{Not Five}}{\overset{\texttt{Five}}{\gtrless}} 0$$

This may be written in terms of a threshold γ,

$$g(\boldsymbol{x}) \underset{\texttt{Not Five}}{\overset{\texttt{Five}}{\gtrless}} \gamma$$

where $\gamma = 0$. For this classifier, there is a certain precision and recall, and if this threshold is changed, then the precision and recall will change. As illustrated in Fig. 4, as the threshold increases, the precision increases whereas the recall decreases. A plot of the precision and the recall versus the threshold is shown in Fig. 5(a). Similarly, if precision is plotted versus recall for different values of the threshold, we will have a plot as shown in Fig. 5(b). Therefore, by varying the threshold, we are able to adjust the classifier to the desired precision and threshold.

The ROC curve is similar to Fig. 5(b), except instead of plotting precision and recall versus the threshold, the ROC plots the **true positive rate** (another name for recall) versus the **false positive rate**, which is one minus the true negative rate. An example of an ROC curve is shown in Fig. 6. The top left corner of the plot is the **ideal** point where the false positive

[7] ROC comes from communications theory and stands for Receiver Operating Characteristic.

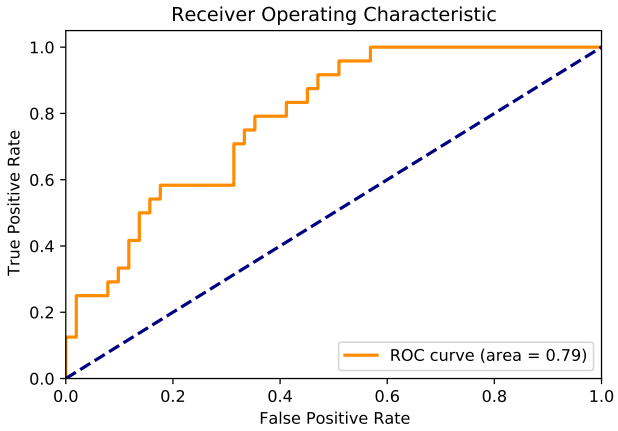


Figure 6: ROC Curve.

rate is zero and a true positive rate is one. Although one cannot expect to achieve this, it does mean that a larger **Area Under the ROC Curve**, (AUC), is usually better. The *steepness* of ROC curves is also important, since it is ideal to maximize the true positive rate while minimizing the false positive rate.

Here is some code to produce an ROC curve (without the legend, titles and labels):

```
from sklearn.metrics import roc_auc_score, roc_curve
roc_score = roc_auc_score(y_test,LR.predict(X_test))
fpr, tpr, thr = roc_curve(y_test, LR.predict_proba(X_test)[:,1])
plt.figure()
plt.plot(fpr,tpr, label='Logistic Regression Area = {:.2f}'.format(roc_score))
plt.plot([0,1],[0,1],'r--')
plt.show()
```

Assignment

(a) Make an ROC plot for your logistic regression classifier.

(b) Where on the ROC curve is our logistic regression classifier when the threshold is zero?

(c) Can you make any arguments for increasing or decreasing the threshold?

(d) What does the diagonal dashed line in the ROC plot represent?

7. Feature Importance

After learning a classifier, it is sometime instructive and informative to look at the coefficients to see which ones have the largest magnitude. When the coefficient w_k multiplying feature x_k in the classifier

$$g(\boldsymbol{x}) = w_0 + w_1 x_1 + \cdots + w_k x_k + \cdots + w_p x_p$$

is small in magnitude, then this feature is not as important in the classification decision as those that have large values because the final classification is based on whether $g(\boldsymbol{x})$ is greater than zero or less than zero.[8] So, to get a sense of what is going on inside the logistic regression model, we can visualize how our model uses the different features and which features have

[8]Note that this assumes that the features have been scaled, otherwise small weights applied to very large feature values may still make a significant contribution.

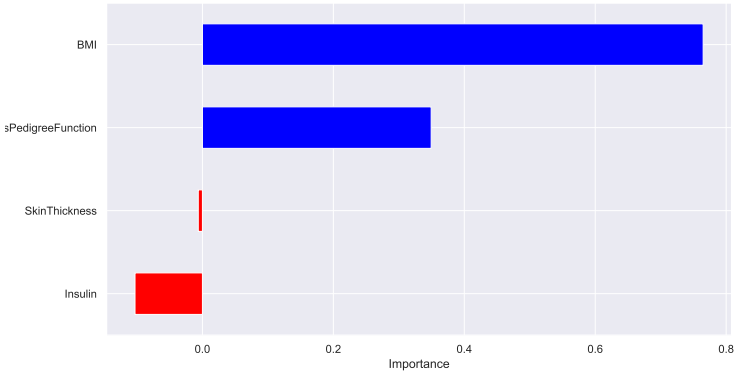


Figure 7: Showing the importance of four features.

greater effect. To do this, we begin by extracting the logistic regression coefficients for each feature and the labels associated with these features,

```
coeff = list(clf.coef_[0])
labels = list(DF_train.drop('Outcome',1).columns)
```

We then create a DataFrame called `features` and in one column called `features` we put the labels, and in the second, called `Importance` we put the coefficients.

```
features = pd.DataFrame()
features['Features'] = labels[4:8]
features['importance'] = coeff[4:8]
```

Now all that is left to do is sort the coefficients and generate a bar chart,

```
features.sort_values(by=['importance'], ascending=True, inplace=True)
features['positive'] = features['importance'] > 0
features.set_index('Features', inplace=True)
features.importance.plot(kind='barh', figsize=(11, 6),
       color = features.positive.map({True: 'blue', False: 'red'}))
plt.xlabel('Importance')
```

An example of what this chart might look like, for four of the features, is shown in Fig. 7. From what is shown in this figure, `BMI` is the most important feature (not to say that the others do not play an important role), and `insulin` has a negative influence on the classifier, which means that with a high value for `insulin`, the chances of having diabetes is reduced.

Experiment

(a) Create a bar chart showing the importance of each of the eight features in the data set.

(b) What are the features that have the most significant influence on the model?

(c) What are the features that a negative effect on diabetes?

(d) Is blood pressure more important as a feature than age?

8. Making Predictions

Having a logistic regression classifier, we are now ready to make predictions. One thing that is interesting to do is to make plots of $\Pr\{Y = 1|\boldsymbol{x}\}$ as a function of a single feature. For example, running logistic regression on only the glucose level of patients,

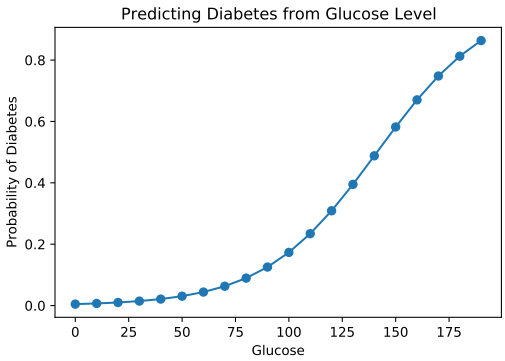


Figure 8: $\Pr\{Y = 1\}|x\}$ where x is the glucose level of a patient.

```
array1 = np.asarray(diabetesDF.iloc[:,1:2])
label1 = np.asarray(diabetesDF['Outcome'])
LR1 = LogisticRegression()
LR1.fit(array1, label1)
```

and plotting the probability versus glucose level,

```
x=np.arange(0,200,10).reshape(-1,1)
y=LR1.predict_proba(x)
plt.scatter(x,y[:,1])
plt.plot(x,y[:,1])
plt.xlabel('Glucose')
plt.ylabel('Probability of Diabetes')
plt.title('Predicting Diabetes from Glucose Level')
```

results in the plot shown in Fig. 8.

Assignment:

(a) What is the probability of having diabetes if you are 25 years old male with a glucose level of 130 and an insulin level of 100, blood pressure of 125, BMI of 32 and skin thickness of 30, and a diabetes pedigree function of 1.1?

(b) Generate a plot of $\Pr\{Y = 1\}|x\}$ where the feature x is the `BMI` and comment on your findings, Does this plot surprise you?

(c) Repeat for a few other features to see if you discover anything interesting.

Extra Credit (Optional)

Produce some plots of $\Pr\{Y = 1\}|x\}$ versus glucose or some other feature for different age groups.