Evaluating a classification model (video #9)

Created by Data School. Watch all 9 videos on YouTube. Download the notebooks from GitHub.

Note: This notebook uses Python 3.6 and scikit-learn 0.19.1. The original notebook (shown in the video) used Python 2.7 and scikit-learn 0.16, and can be downloaded from the archive branch.

Agenda

- What is the purpose of **model evaluation**, and what are some common evaluation procedures?
- What is the usage of **classification accuracy**, and what are its limitations?
- How does a **confusion matrix** describe the performance of a classifier?
- What **metrics** can be computed from a confusion matrix?
- How can you adjust classifier performance by **changing the classification threshold**?
- What is the purpose of an **ROC curve**?
- How does **Area Under the Curve (AUC)** differ from classification accuracy?

Classification accuracy

Pima Indians Diabetes dataset originally from the UCI Machine Learning Repository

In [ ]:
```
# read the data into a pandas DataFrame
import pandas as pd
path = 'data/pima-indians-diabetes.data'
col_names = ['pregnant', 'glucose', 'bp', 'skin', 'insulin', 'bmi', 'pedigre
pima = pd.read_csv(path, header=None, names=col_names)
```

In [ ]:
```
# print the first 5 rows of data
pima.head()
```

Out[ ]:

	pregnant	glucose	bp	skin	insulin	bmi	pedigree	age	label
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Question: Can we predict the diabetes status of a patient given their health measurements?

In [ ]:
```
# define X and y
feature_cols = ['pregnant', 'insulin', 'bmi', 'age']
X = pima[feature_cols]
y = pima.label
```

In [ ]:
```
# split X and y into training and testing sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

In [ ]:
```
# train a logistic regression model on the training set
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression()
logreg.fit(X_train, y_train)
```

Out[ ]:
```
▼ LogisticRegression
LogisticRegression()
```

In [ ]:
```
# make class predictions for the testing set
y_pred_class = logreg.predict(X_test)
```

Classification accuracy: percentage of correct predictions

In [ ]:
```
# calculate accuracy
from sklearn import metrics
print(metrics.accuracy_score(y_test, y_pred_class))
```

```
0.6770833333333334
```

Null accuracy: accuracy that could be achieved by always predicting the most frequent class

In [ ]:
```
# examine the class distribution of the testing set (using a Pandas Series m
y_test.value_counts()
```

Out[ ]:
```
0    130
1     62
Name: label, dtype: int64
```

In [ ]:
```
# calculate the percentage of ones
y_test.mean()
```

Out[ ]:
```
0.3229166666666667
```

In [ ]:
```
# calculate the percentage of zeros
1 - y_test.mean()
```

Out[ ]:
```
0.6770833333333333
```

In [ ]:
```
# calculate null accuracy (for binary classification problems coded as 0/1)
```

```
max(y_test.mean(), 1 - y_test.mean())
```

Out[ ]: 0.6770833333333333

In [ ]:
```
# calculate null accuracy (for multi-class classification problems)
y_test.value_counts().head(1) / len(y_test)
```

Out[ ]:
```
0    0.677083
Name: label, dtype: float64
```

Comparing the **true** and **predicted** response values

In [ ]:
```
# print the first 25 true and predicted responses
print('True:', y_test.values[0:25])
print('Pred:', y_pred_class[0:25])
```

```
True: [1 0 0 1 0 0 1 1 0 0 1 1 0 0 0 0 1 0 0 0 1 1 0 0 0]
Pred: [0 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

Conclusion:

- Classification accuracy is the **easiest classification metric to understand**
- But, it does not tell you the **underlying distribution** of response values
- And, it does not tell you what **"types" of errors** your classifier is making

Confusion matrix

Table that describes the performance of a classification model

In [ ]:
```
# IMPORTANT: first argument is true values, second argument is predicted val
print(metrics.confusion_matrix(y_test, y_pred_class))
```

```
[[114  16]
 [ 46  16]]
```

n=192	**Predicted: 0**	**Predicted: 1**
Actual: 0	118	12
Actual: 1	47	15

- Every observation in the testing set is represented in **exactly one box**
- It's a 2x2 matrix because there are **2 response classes**

- The format shown here is **not** universal

Basic terminology

- **True Positives (TP):** we *correctly* predicted that they *do* have diabetes
- **True Negatives (TN):** we *correctly* predicted that they *don't* have diabetes
- **False Positives (FP):** we *incorrectly* predicted that they *do* have diabetes (a "Type I error")
- **False Negatives (FN):** we *incorrectly* predicted that they *don't* have diabetes (a "Type II error")

In [ ]:
```
# print the first 25 true and predicted responses
print('True:', y_test.values[0:25])
print('Pred:', y_pred_class[0:25])
```

```
True: [1 0 0 1 0 0 1 1 0 0 1 1 0 0 0 1 0 0 0 1 1 0 0 0]
Pred: [0 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
```

In [ ]:
```
# save confusion matrix and slice into four pieces
confusion = metrics.confusion_matrix(y_test, y_pred_class)
TP = confusion[1, 1]
TN = confusion[0, 0]
FP = confusion[0, 1]
FN = confusion[1, 0]
```

n=192	**Predicted: 0**	**Predicted: 1**	
Actual: 0	TN = 118	FP = 12	130
Actual: 1	FN = 47	TP = 15	62
	165	27	

Metrics computed from a confusion matrix

Classification Accuracy: Overall, how often is the classifier correct?

In [ ]:
```
print((TP + TN) / float(TP + TN + FP + FN))
print(metrics.accuracy_score(y_test, y_pred_class))
```

```
0.6770833333333334
0.6770833333333334
```

Classification Error: Overall, how often is the classifier incorrect?

- Also known as "Misclassification Rate"

In [ ]:
```
print((FP + FN) / float(TP + TN + FP + FN))
print(1 - metrics.accuracy_score(y_test, y_pred_class))
```

```
0.3229166666666667
0.32291666666666663
```

Sensitivity: When the actual value is positive, how often is the prediction correct?

- How "sensitive" is the classifier to detecting positive instances?
- Also known as "True Positive Rate" or "Recall"

In [ ]:
```
print(TP / float(TP + FN))
print(metrics.recall_score(y_test, y_pred_class))
```

```
0.25806451612903225
0.25806451612903225
```

Specificity: When the actual value is negative, how often is the prediction correct?

- How "specific" (or "selective") is the classifier in predicting positive instances?

In [ ]:
```
print(TN / float(TN + FP))
```

```
0.8769230769230769
```

False Positive Rate: When the actual value is negative, how often is the prediction incorrect?

In [ ]:
```
print(FP / float(TN + FP))
```

```
0.12307692307692308
```

Precision: When a positive value is predicted, how often is the prediction correct?

- How "precise" is the classifier when predicting positive instances?

In [ ]:
```
print(TP / float(TP + FP))
print(metrics.precision_score(y_test, y_pred_class))
```

```
0.5
0.5
```

Many other metrics can be computed: F1 score, Matthews correlation coefficient, etc.

Conclusion:

- Confusion matrix gives you a **more complete picture** of how your classifier is performing
- Also allows you to compute various **classification metrics**, and these metrics can guide your model selection

Which metrics should you focus on?

- Choice of metric depends on your **business objective**
- **Spam filter** (positive class is "spam"): Optimize for **precision or specificity** because false negatives (spam goes to the inbox) are more acceptable than false positives (non-spam is caught by the spam filter)
- **Fraudulent transaction detector** (positive class is "fraud"): Optimize for **sensitivity** because false positives (normal transactions that are flagged as possible fraud) are more acceptable than false negatives (fraudulent transactions that are not detected)

Adjusting the classification threshold

In [ ]:
```
# print the first 10 predicted responses
logreg.predict(X_test)[0:10]
```

Out[ ]:
```
array([0, 0, 0, 0, 0, 0, 0, 1, 0, 1])
```

In [ ]:
```
# print the first 10 predicted probabilities of class membership
logreg.predict_proba(X_test)[0:10, :]
```

Out[ ]:
```
array([[0.61405867, 0.38594133],
       [0.7505398 , 0.2494602 ],
       [0.74167648, 0.25832352],
       [0.60291327, 0.39708673],
       [0.88426611, 0.11573389],
       [0.87695895, 0.12304105],
       [0.50819992, 0.49180008],
       [0.44582289, 0.55417711],
       [0.77950769, 0.22049231],
       [0.25853303, 0.74146697]])
```

In [ ]:
```
# print the first 10 predicted probabilities for class 1
logreg.predict_proba(X_test)[0:10, 1]
```

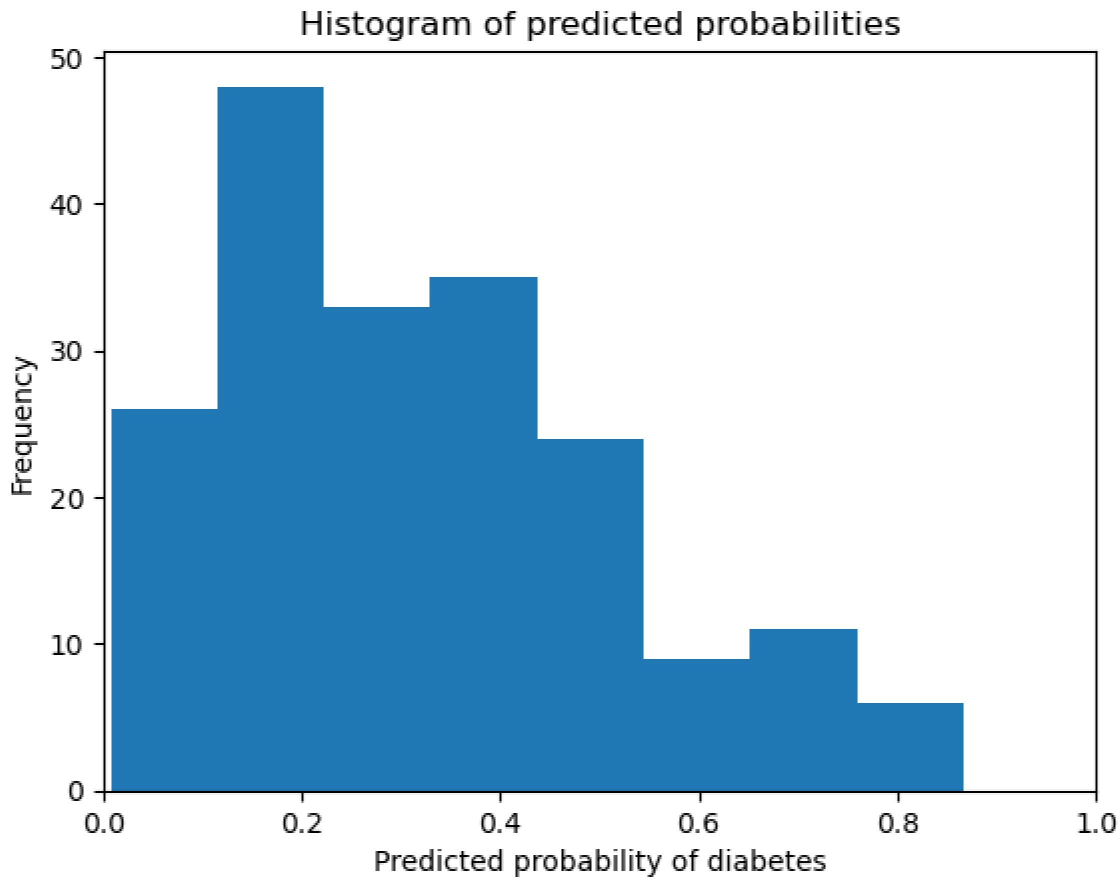
Out[ ]:
```
array([0.38594133, 0.2494602 , 0.25832352, 0.39708673, 0.11573389,
       0.12304105, 0.49180008, 0.55417711, 0.22049231, 0.74146697])
```

In [ ]:
```
# store the predicted probabilities for class 1
y_pred_prob = logreg.predict_proba(X_test)[:, 1]
```

In [ ]:
```
# allow plots to appear in the notebook
%matplotlib inline
import matplotlib.pyplot as plt
```

In [ ]:
```
# histogram of predicted probabilities
plt.hist(y_pred_prob, bins=8)
plt.xlim(0, 1)
plt.title('Histogram of predicted probabilities')
plt.xlabel('Predicted probability of diabetes')
plt.ylabel('Frequency')
```

Out[ ]:
```
Text(0, 0.5, 'Frequency')
```



Decrease the threshold for predicting diabetes in order to **increase the sensitivity** of the classifier

In [ ]:
```
# predict diabetes if the predicted probability is greater than 0.3
from sklearn.preprocessing import binarize
# Added positional parameter name "threshold"
y_pred_class = binarize([y_pred_prob], threshold=0.3)[0]
```

In [ ]:
```
# print the first 10 predicted probabilities
y_pred_prob[0:10]
```

Out[ ]:
```
array([0.38594133, 0.2494602 , 0.25832352, 0.39708673, 0.11573389,
       0.12304105, 0.49180008, 0.55417711, 0.22049231, 0.74146697])
```

In [ ]:
```
# print the first 10 predicted classes with the lower threshold
y_pred_class[0:10]
```

Out[ ]:
```
array([1., 0., 0., 1., 0., 0., 1., 1., 0., 1.])
```

In [ ]:
```
# previous confusion matrix (default threshold of 0.5)
print(confusion)
```

```
[[114  16]
 [ 46  16]]
```

In [ ]:
```
# new confusion matrix (threshold of 0.3)
print(metrics.confusion_matrix(y_test, y_pred_class))
```

```
[[82 48]
 [17 45]]
```

In [ ]:
```
# sensitivity has increased (used to be 0.24)
print(46 / float(46 + 16))
```

```
0.7419354838709677
```

In [ ]:
```
# specificity has decreased (used to be 0.91)
print(80 / float(80 + 50))
```

```
0.6153846153846154
```

Conclusion:

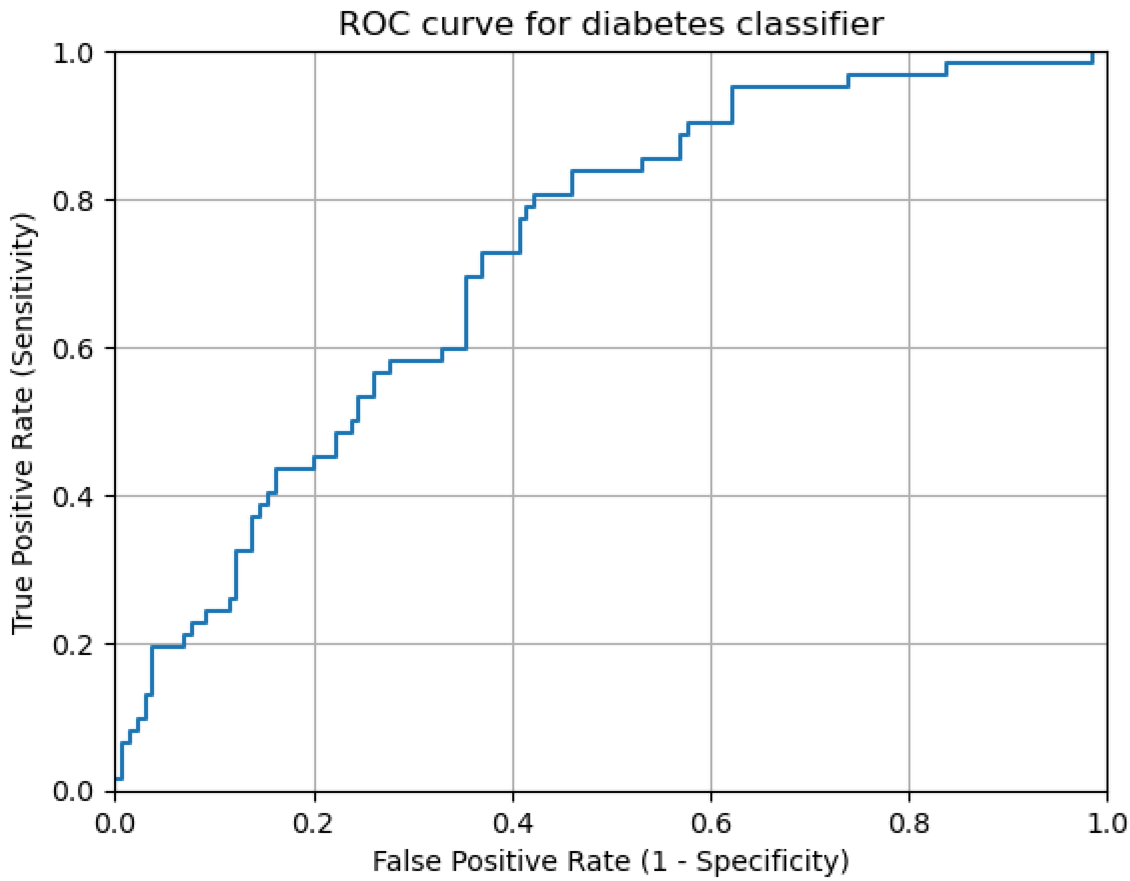
- **Threshold of 0.5** is used by default (for binary problems) to convert predicted probabilities into class predictions
- Threshold can be **adjusted** to increase sensitivity or specificity
- Sensitivity and specificity have an **inverse relationship**

ROC Curves and Area Under the Curve (AUC)

Question: Wouldn't it be nice if we could see how sensitivity and specificity are affected by various thresholds, without actually changing the threshold?

Answer: Plot the ROC curve!

In [ ]:
```
# IMPORTANT: first argument is true values, second argument is predicted pro
fpr, tpr, thresholds = metrics.roc_curve(y_test, y_pred_prob)
plt.plot(fpr, tpr)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.0])
plt.title('ROC curve for diabetes classifier')
plt.xlabel('False Positive Rate (1 - Specificity)')
plt.ylabel('True Positive Rate (Sensitivity)')
plt.grid(True)
```



- ROC curve can help you to **choose a threshold** that balances sensitivity and specificity in a way that makes sense for your particular context
- You can't actually **see the thresholds** used to generate the curve on the ROC curve itself

In [ ]:
```
# define a function that accepts a threshold and prints sensitivity and spec
def evaluate_threshold(threshold):
    print('Sensitivity:', tpr[thresholds > threshold][-1])
    print('Specificity:', 1 - fpr[thresholds > threshold][-1])
```

In [ ]:
```
evaluate_threshold(0.5)
```

```
Sensitivity: 0.25806451612903225
Specificity: 0.8769230769230769
```

In [ ]:
```
evaluate_threshold(0.3)
```

```
Sensitivity: 0.7258064516129032
Specificity: 0.6307692307692307
```

AUC is the **percentage** of the ROC plot that is **underneath the curve**:

In [ ]:
```
# IMPORTANT: first argument is true values, second argument is predicted pro
print(metrics.roc_auc_score(y_test, y_pred_prob))
```

```
0.7227047146401985
```

- AUC is useful as a **single number summary** of classifier performance.
- If you randomly chose one positive and one negative observation, AUC represents the likelihood that your classifier will assign a **higher predicted probability** to the positive observation.
- AUC is useful even when there is **high class imbalance** (unlike classification accuracy).

In [ ]:
```
# calculate cross-validated AUC
from sklearn.model_selection import cross_val_score
cross_val_score(logreg, X, y, cv=10, scoring='roc_auc').mean()
```

Out[ ]:
```
0.7425071225071225
```

Confusion matrix advantages:

- Allows you to calculate a **variety of metrics**
- Useful for **multi-class problems** (more than two response classes)

ROC/AUC advantages:

- Does not require you to **set a classification threshold**
- Still useful when there is **high class imbalance**

Confusion Matrix Resources

- Blog post: Simple guide to confusion matrix terminology by me
- Videos: Intuitive sensitivity and specificity (9 minutes) and The tradeoff between sensitivity and specificity (13 minutes) by Rahul Patwari
- Notebook: How to calculate "expected value" from a confusion matrix by treating it as a cost-benefit matrix (by Ed Podojil)
- Graphic: How classification threshold affects different evaluation metrics (from a blog post about Amazon Machine Learning)

ROC and AUC Resources

- Video: ROC Curves and Area Under the Curve (14 minutes) by me, including transcript and screenshots and a visualization
- Video: ROC Curves (12 minutes) by Rahul Patwari
- Paper: An introduction to ROC analysis by Tom Fawcett
- Usage examples: Comparing different feature sets for detecting fraudulent Skype users, and comparing different classifiers on a number of popular datasets

Other Resources

- scikit-learn documentation: Model evaluation
- Guide: Comparing model evaluation procedures and metrics by me
- Video: Counterfactual evaluation of machine learning models (45 minutes) about how Stripe evaluates its fraud detection model, including slides

Comments or Questions?

- Email: kevin@dataschool.io
- Website: http://dataschool.io
- Twitter: @justmarkham

In [ ]:

```
from IPython.core.display import HTML
def css_styling():
    styles = open("styles/custom.css", "r").read()
    return HTML(styles)
css_styling()
```

Out[ ]: