CS 236756 - Technion - Intro to Machine Learning

Tal Daniel

Tutorial 05 - Evaluation & Cross-Validation

Agenda

- Motivation
- Metrics for Classifier's Evaluation
- Methods for Classifier's Evaluation
  - Holdout
  - K-Fold Cross-Validation
  - Stratification
  - Leave-one-out

Motivation - Why Evaluate Classifier's Generalization Ability?

It is important to evaluate the classifier generalization performance in order to:

- Determine whether to employ/distribute the classifier
- Compare classifiers (even compare the same type of classifiers, but with different parameters)
- Optimize the classifier to perform better on unseen data

What Do We Need To Do That?

The first question the needs to be asked, is what preparations are needed in order to evaluate a trained model.

- **Train-Test Separation** - The *naive* approach is seprating the data into train set and test set, that is, taking a portion of the data for training the model (usually about 80% of the dataset) and save another portion, that the model **has not seen** in order to test the model's performance. This is called the test set (usually about 20% of the dataset).
- Note: Scikit-learn has a function we can use called `train_test_split` that makes it easy for us to split our dataset into training and testing data.

How do we make sure the separation is fair and that each set (train and test) is a good representation of the data distribution?

- **Shuffling** - Shuffling the data serves the purpose of reducing variance and making sure that models remain general and overfit less. The popular case where shuffling is very important is when the data is sorted by their class/target. By shuffling we add randomness that assures that the training/test/validation sets are representative of the overall distribution of the data.

In [1]:
```
# imports for the tutorial
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib notebook
```

In [5]:
```
# let's load the cancer dataset, shuffle it and  speratre into train and test set
dataset = pd.read_csv('./datasets/cancer_dataset.csv')
# print the number of rows in the data set
number_of_rows = len(dataset)
print("total samples: {}".format(number_of_rows))
total_positive_samples = np.sum(dataset['diagnosis'].values == 'M')
print("total positive sampels (M): {}, total negative samples (B): {}".format(total_positive_samples, numb
er_of_rows - total_positive_samples))
num_train = int(0.8 * number_of_rows)
# reminder, the data looks like this
# dataset.head(10) # the dataset is ordered by the diagnosis
dataset.sample(10)
```

```
total samples: 569
total positive sampels (M): 212, total negative samples (B): 357
```

Out[5]:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_me
246	884448	B	13.20	17.43	84.13	541.6	0.07215	0.04524
272	8910988	M	21.75	20.99	147.30	1491.0	0.09401	0.19610
355	9010258	B	12.56	19.07	81.92	485.8	0.08760	0.10380
403	9047	B	12.94	16.17	83.18	507.6	0.09879	0.08836
125	86561	B	13.85	17.21	88.44	588.7	0.08785	0.06136
377	9013579	B	13.46	28.21	85.89	562.1	0.07517	0.04726
501	91504	M	13.82	24.49	92.33	595.9	0.11620	0.16810
284	8912284	B	12.89	15.70	84.08	516.6	0.07818	0.09580
431	907915	B	12.40	17.68	81.47	467.8	0.10540	0.13160
438	909231	B	13.85	19.60	88.68	592.6	0.08684	0.06330

10 rows × 33 columns

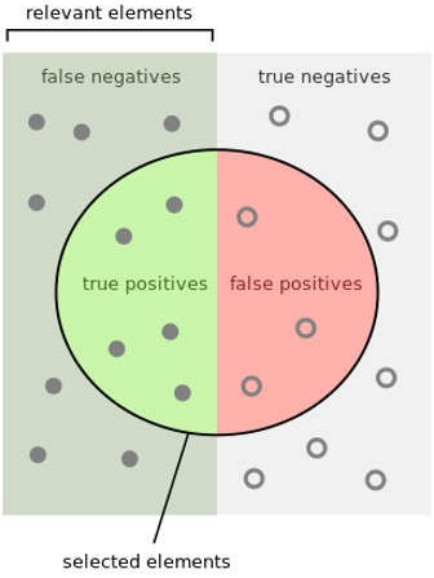
In [3]:
```
# we will take the first 2 features as our data (X) and the diagnosis as labels (y)
x = dataset[['radius_mean', 'texture_mean']].values
y = dataset['diagnosis'].values == 'M'  # 1 for Malignant, 0 for Benign
# shuffle
rand_gen = np.random.RandomState(0)
shuffled_indices = rand_gen.permutation(np.arange(len(x)))

x_train = x[shuffled_indices[:num_train]]
y_train = y[shuffled_indices[:num_train]]
x_test = x[shuffled_indices[num_train:]]
y_test = y[shuffled_indices[num_train:]]

print("total training samples: {}, total test samples: {}".format(num_train, number_of_rows - num_train))
```

```
total training samples: 455, total test samples: 114
```

Metrics for Classifier's Evaluation

Different ML tasks use different metrics to measure the models' performance, we will introduce several of them, but there are more, and sometimes there are tasks that require hand-crafted metrics (for example, an NLP tasks of translating sentences from Chinese to English is measured by the BLEU score, which is a linguistic measure of language coherence).

Throught this part, we will use the terms: True Positive (TP), False Positive (FP), True Negative (TN), False Negative (FN) which are demonstrated below:



- **Accuracy** - $\frac{TP+TN}{P+N}$
  - In simple words: how many did we get *right* out of all of the dataset?
  - When not to use *accuracy*?
    - When dealing with *skewed* datasets (i.e., when some classes/labels are more frequent than others).
- **Error** - $\frac{FP+FN}{P+N}$
  - In simple words: how many did we get *wrong* out of all of the dataset?
- **Precision** - $\frac{TP}{TP+FP}$
  - In simple words: out of all the samples we calssified as *positive*, how many of them we got *right*. The accuracy of positive predictions.
  - **Always** calculated along with **Recall**
- **Recall (TP Rate, Sensitivity)** - $\frac{TP}{P} = \frac{TP}{TP+FN}$
  - In simple words: out of all the *positive* samples, how many of them we got *right*.
  - **Always** calculated along with **Precision**
- **FP Rate** - $\frac{FP}{N} = \frac{FP}{FP+TN}$
  - In simple words: out of all the *negative* samples, how many of them we got *wrong*, or, the ratio of negative instances that are incorrectly classified as positive.

Quick Examples

- A classifier that is trained to detect videos that are safe for kids. You would probably prefer a classifier that rejects many good videos (low recall) but keeps only safe ones (high precision), rather than a classifier that has a much higher recall but lets a few really bad videos.
- A classifier trained to detect shoplifters on surveillance images: it is probably fine if your classifier has only 30% precision as long as it has 99% recall (almost all shoplifters will get caught. but a lot of false alarms).

Example - Naive Classifier Evaluation on the Breast Cancer Dataset

Let's create a classifier that classifies new samples by the label probability it has seen in the train set. That is, if it has seen 30% Malignant labels, then with probability 0.3 a new sample is classified as malignant.

We will evaluate using the above metrics.

In [4]:
```
# probability to tag as malignat
m_prob = np.sum(y_train) / len(y_train)
# probability to tag as benign
b_prob = 1 - m_prob
print("M prob: {:.3f}, B prob: {:.3f}".format(m_prob, b_prob))

# now let's classify the test set
# since we don't look at the data at all we can just randomly sample Ms and Bs in the size of the test set
y_test_pred = np.random.choice([True, False], size=(len(y_test)), p=[m_prob, b_prob])

# let's evaluate
accuracy = np.sum(y_test == y_test_pred) / len(y_test)
print("accuracy: {:.3f} or {:.3f} %".format(accuracy, accuracy * 100))
error = np.sum(y_test != y_test_pred) / len(y_test)
print("error: {:.3f} or {:.3f} %".format(error, error * 100))
precision = np.sum(y_test_pred[y_test]) / np.sum(y_test_pred)
print("precision: {:.3f} or {:.3f} %".format(precision, precision * 100))
recall = np.sum(y_test_pred[y_test]) / np.sum(y_test)
print("recall: {:.3f} or {:.3f} %".format(recall, recall * 100))
fp_rate =  np.sum(y_test_pred[~y_test]) / np.sum(~y_test)
print("FP Rate: {:.3f} or {:.3f} %".format(fp_rate, fp_rate * 100))
```

```
M prob: 0.369, B prob: 0.631
accuracy: 0.553 or 55.263 %
error: 0.447 or 44.737 %
precision: 0.405 or 40.541 %
recall: 0.341 or 34.091 %
FP Rate: 0.314 or 31.429 %
```

In [5]:
```
# using scikit-learn
from sklearn.metrics import precision_score, recall_score
precision = precision_score(y_test, y_test_pred)
print("precision: {:.3f} or {:.3f} %".format(precision, precision * 100))
recall = recall_score(y_test, y_test_pred)
print("recall: {:.3f} or {:.3f} %".format(recall, recall * 100))
```

```
precision: 0.405 or 40.541 %
recall: 0.341 or 34.091 %
```

Confusion Matrix Revisited

A better way to evaluate the performance of a calssifier is the look at the *confusion matrix*. Each row in a confusion matrix represents an actual calss. A perfect classifier would have only true positives and true negatives, so its confusion matrix would have non-zeros values only on its main diagonal.

		Predicted Class	
		Pos	Neg
Actual Class	Pos	*TP*	*FN*
	Neg	*FP*	*TN*

In [6]:
```
# using scikit-learn
from sklearn.metrics import confusion_matrix
conf_mat = confusion_matrix(y_test, y_test_pred)
conf_mat_df = pd.DataFrame(conf_mat, columns=['Predicted Pos', 'Predicted Neg'], index=['Actual Pos', 'Act
ual Neg'])
conf_mat_df
```

Out[6]:

	Predicted Pos	Predicted Neg
Actual Pos	48	22
Actual Neg	29	15

Precision/Recall Tradeoff

Increasing precision reduces recall, and vice versa. It is often convenient to combine them into a single metric called the F_1 score.

The F_1 Score

It is the *harmonic mean* of precision and recall.

$$F_1 = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} = 2 \times \frac{precision \times recall}{precision + recall} = \frac{TP}{TP + \frac{FN+FP}{2}}$$

Whereas the regular mean treats all values equally , the harmonic mean gives much more weight to **low values.** As a result, the classifier will only get a high F_1 score if both recall and precision are high.

The F_1 score favors classifiers that have similar precision and recall. This is not always what we want, as seen in the "Quick Examples" above.

In [7]:
```
# calculating the f1 score
f_1_score = 2 * precision * recall / (precision + recall)
print("f1 score: {:.3f}".format(f_1_score))
# using scikit-learn
from sklearn.metrics import f1_score
f_1_score = f1_score(y_test, y_test_pred)
print("f1 score (scikit): {:.3f}".format(f_1_score))
```

```
f1 score: 0.370
f1 score (scikit): 0.370
```

Methods for Classifier's Evaluation

So far we have separated our data to a train set and *test set* to evaluate our classifier. This is also called **hold-out** method.

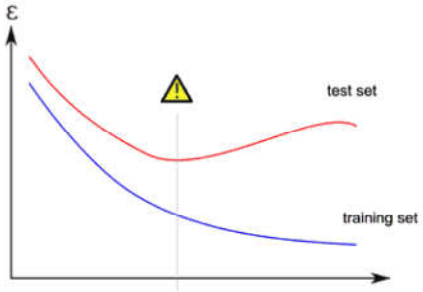
Let's list the ways we can estimate the metrics for a certain classifier:

- **Training Data**
- **Independent Test Data** (different from hold-out)
- **Hold-Out Method**
- K**-fold Cross-Validation**
- **Leave-One-Out Method**
- **Bootstrap Method**
- And more...

We will now present some of them that you will use.

Estimation with Training Data

- The accuracy/error estimates ont the *training* data **are not** good indicators of performance on future data!
- The reason: new data will probably not be exactly the same as the training data.
- The accuracy/error estimates on the training data measure the degree of calssifier's **underfitting** or **overfitting**.
- A typical learning curve usually looks like this:

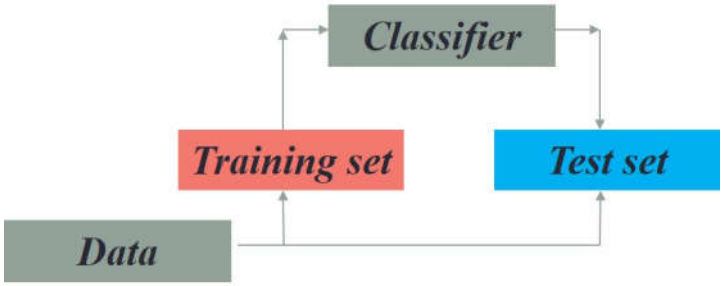


(image from mlwiki.org (http://mlwiki.org/index.php/Overfitting))

Estimation with Independent Test Data

- Estimation with independent test data is used when we have plenty of data and there is **a natural way to generating this data**, that is, there is no need to separate the the train data.
- For example, we wish to learn a certainn function, like $f(x) = x^2$, we can generate how many samples we want and we are not bounded to a certain dataset.
- In most ML tasks, we have no natural way of forming data, and we are limited to the data given to us, which is why it is more common to use hold-out.

Hold-Out Method



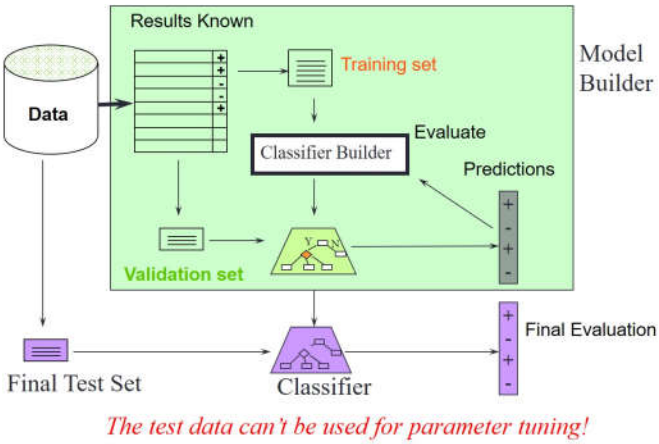
- The hold-out method **splits** the data into training data and test data. Then, we build a classifier using the train set and test it using the test set.
- The hold-out method is usually used when we have thousands of instances, including several hundred instances from each class.
- Scikit-learn has a built in function that you can use to split the data: `from sklearn.model_selection import train_test_split`

Train-Validation-Test Split

Let's say you have a dataset which you separated into *train* set and *test* set. You want to train a classifier that has hyper-parameters (parameters that are not trained, but are user-selected before the training) that you have to tune. For example, in the classifier "K-Nearest Neighbours" you need to choose K, the number of nearest neighbours needed to perform classification, or in "Stochastic Gradient Descent", you need to choose the learning rate (which is a continuous value, like 0.0001). Would it be fair to train the classifier on the train set for each hyper-parameter and then test it on the test set and finally selcting the best hyper-parameters based on the performance on the test set? **NO!**

It is like taking an open-material exam, but instead of bringing all of the material, you bring only the material relevant to the questions asked in the exam. It is sort of cheating. That is why we seperate into 3 sets:

- **Tran Set** - from which the model learns
- **Validation Set** - on which the hyper-parameters are tuned
- **Test Set** - untouched samples on which you test the generalization ability of the model. This set has **never** been seen by the model.



Making the Most of the Data

- Once evaluation is complete and you are satisfied with the model, *all* the data can be used to build the final classifier
- Generally, the **larger the training data** the **better the classifier**.
- The **larger the test set** the **more accurate** the error estimate.

Stratification

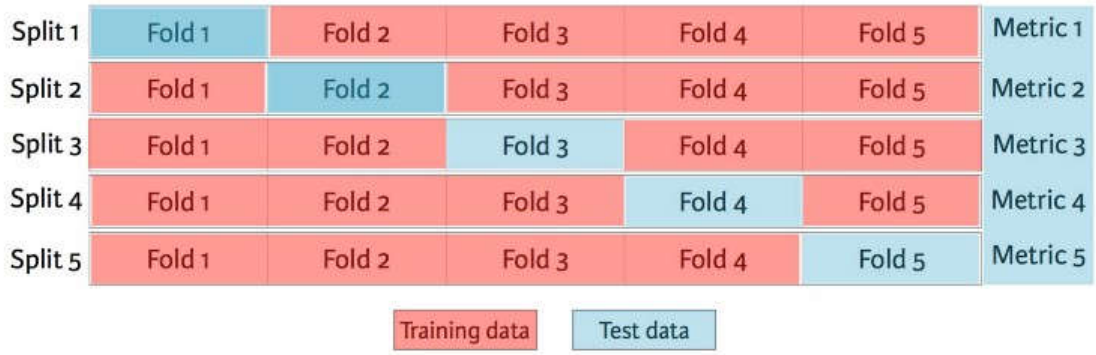
- The *holdout* method reserves a certain amount for testing and uses the remainder for training.
- For **"unbalanced"/skewed** datasets, samples might not be representative.
  - Few or None instances of some classes
- **Stratified Sampling**: balancing the data. If the dataset is not large enough, there is a risk of introducing a significant sampling *bias*. In *stratified sampling*, the population is divided into homogenous subgroups called *strata*, and the right number of instances is sampled from each stratum to guarantee that the test set is representative of the overall population.
  - Makes sure that each class is represented with approximately equal proportions in both subsets.
- For example, when a survey company decides to call 1,000 people to ask them a few questions, they don't just pick up randomly 1,000 people from a phone book. They try to ensure that these 1,000 people are representative of the whole population. In the US, the population is composed of 51.3% female and 48.7% men, so a well-conducted survey in the US would try to maintain this ratio in the samples - 513 female and 487 male.

- In Scikit-learn - `from sklearn.model_selection import StratifiedShuffleSplit` - Read More Here (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedShuffleSplit.html)

K-Fold Cross-Validation

Seprating to validation and test sets is not very data efficient as we now allocate less data for training. In order to make better use of the data we use a technique called "Cross Validation". However, in this course we will always do this sepration regardless! (we'll use the train set for CV). Cross-validation is randomly splitting the data into k groups. One of the groups is used as the test set and the rest are used as the training set. The model is trained on the training set and scored on the test set. Then the process is repeated until each unique group as been used as the test set. This method is also called "K-Fold Cross Validation".

For example, for 5-fold cross validation, the dataset is split into 5 groups, and the model is trained and tested 5 separate times so each group gets a chance to be the test set. This can be seen in the image below.



(image from Datacamp)

Cross-validation is better than using the holdout method because the holdout method score is dependent on how the data is split into train, validation and test sets. Cross-validation gives the model an opportunity to test on multiple splits so we can get a better idea on how the model will perform on unseen data.

The steps:

1. Data is split into k subsets of equal size.
2. Each subset in turn is used for testing and the reaminder for training.
3. The estimates are averaged to yield an overall estimate.

Implementation using Scikit-learn:

- `cross_val_score` - takes in a classifier, training data, k (number of folds) and the scoring technique ("accuracy" for example). Read the Doc Here (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html)
- `StratifiedKFold` - if you need more control over the cross-validation process than provided in `cross_val_score`, it is possible to implement cross-validation yourself. The `StratifiedKFold` class performs stratified sampling to produce folds that contain a representative ratio of each class. Read the Doc Here (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html)

More on Cross-Validation:

- Standard method for evaluation: *stratified 10-fold cross validation*
- Stratification reduces the estimate's *variance* (more confidence in the output)
- An even better method: **repeated** stratified cross-validation
  - For example, 10-fold cross-validation is repeated 10 times and the results are averaged (reduces the variance).
- Which dataset should we use for the K-Fold Cross Validation (Train, Test, Validation)?
  - The **test set** - must be untouched, put aside. Don't use it for ANY purpose other than testing.
  - The **validation set** - used to test performances of the models at various stages of the model development (like tuning the hyper-parameters of the model)
  - The **train set** - used for training, including cross-validation. USE ONLY THIS FOR CV!

Leave-One-Out Cross-Validation

- A particular form of cross-validation where the number of folds = number of training instances
  - That is, for training set of size n, the classifier is built n times
- Makes best use of the data
- Involves **no random subsampling**
- Very computationally **expensive**
- **Stratification is not possible** - it is a big disadvantage
  - It *guarantees* a non-stratified sample because there is only one instance in the test set.

- Extreme example - random datatset split equally into two classes:
  - Best inducer predicts majority class
  - 50% accuracy on fresh data
  - Leave-One-Out CV estimate is 100% error!

Pre-proceesing steps on validation steps

- Whatever data preparations steps done on the training data, should be able to be applied at the final test on unseen data.
  - For example, if normalizing a feature with a mean, the mean is fixed at the data prep and **not recalculated**. Otherwise, a model that was trained and tested on data that used the former mean may not be adequate.
- Data preparation steps are mostly based on descriptive statistics. So a **larger sample** that provides a **stable statistics** is an advantage.
- By not using the validation set, we have a **better estimate of the "true error"**, which we measure by applying pre-processing steps to test data.
- By using the validation set, we **separate the effect of the pre-processing** from that of the classifier.

🦸 Classification Example - Breast Cancer - From Zero to Hero

We will now demonstrate all that we have learned on the Breast Cancer dataset. For the purpose of the exercise, we will use a classifier called "SGDClassifier" (SVM using Stochastic Gradient Descent). You are not supposed to be familiar with it at this point of the course, you can read more about it (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html). We will use it as a black-box algorithm that gives us classification for a certain dataset.

- We will only use *train* and *test* sets since the dataset has limited number of samples and since we are not going to tune the hyper-parameters of the model.
- Each classifier has a **decision function** - for each instance, it computes a score and if that score is greater than some threshold, it assigns the instance to the positive class, or else it assigns it to the negative class (lowering the threshold usually leads to increase in *recall* and reduce in *precision*, and vice versa.
- Finally, will briefly introduce the **ROC Curve**, as a tool to asses to performance of the classifier.

In [8]:
```
# scikit-learn imports
from sklearn.model_selection import cross_val_score, cross_val_predict
from sklearn.metrics import confusion_matrix, f1_score, precision_recall_curve, roc_curve, roc_auc_score
from sklearn.linear_model import SGDClassifier
from sklearn.preprocessing import StandardScaler
```

In [9]:
```
# prepare the dataset
# we will take the first 2 features as our data (X) and the diagnosis as labels (y)
x = dataset[['radius_mean', 'texture_mean']].values
y = dataset['diagnosis'].values == 'M'  # 1 for Malignat, 0 for Benign
# x = scaler.fit_transform(x)
# shuffle
rand_gen = np.random.RandomState(0)
shuffled_indices = rand_gen.permutation(np.arange(len(x)))

x_train = x[shuffled_indices[:num_train]]
y_train = y[shuffled_indices[:num_train]]
x_test = x[shuffled_indices[num_train:]]
y_test = y[shuffled_indices[num_train:]]

# pre-process - standartization
scaler = StandardScaler()
scaler.fit(x_train)
x_train = scaler.transform(x_train)
x_test = scaler.transform(x_test)

print("total training samples: {}, total test samples: {}".format(num_train, number_of_rows - num_train))
```

```
total training samples: 455, total test samples: 114
```

In [10]:
```
# create the classifier
sgd_clf = SGDClassifier(random_state=92, max_iter=1000, tol=1e-3)  # we make the random state constant for
reproducible results
# train (fit) using cross validation
k_folds = 10
cross_val_scores = cross_val_score(sgd_clf, x_train, y_train, cv=k_folds, scoring='accuracy')
print("accuracy in each fold:")
print(cross_val_scores)
print("mean training accuracy:")
print(cross_val_scores.mean())

# Now, fit the classifier to the train data
sgd_clf.fit(x_train, y_train)
```

```
accuracy in each fold:
[0.82608696 0.86956522 0.84782609 0.91304348 0.89130435 0.82608696
 0.91304348 0.82222222 0.88636364 0.88636364]
mean training accuracy:
0.8681906016688625
```

Out[10]:
```
SGDClassifier(alpha=0.0001, average=False, class_weight=None, epsilon=0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.15,
       learning_rate='optimal', loss='hinge', max_iter=1000, n_iter=None,
       n_jobs=1, penalty='l2', power_t=0.5, random_state=92, shuffle=True,
       tol=0.001, verbose=0, warm_start=False)
```

In [11]:
```
# evaluation
y_test_pred = sgd_clf.predict(x_test)
# confusion matrix
conf_mat = confusion_matrix(y_test, y_test_pred)
conf_mat_df = pd.DataFrame(conf_mat, columns=['Predicted Pos', 'Predicted Neg'], index=['Actual Pos', 'Act
ual Neg'])
conf_mat_df
```

Out[11]:

	Predicted Pos	Predicted Neg
Actual Pos	62	8
Actual Neg	4	40

In [12]:
```
# accuracy
accuracy = np.sum(y_test == y_test_pred) / len(y_test)
# f1 score
f1 = f1_score(y_test, y_test_pred)
print("accuracy: {:.3f} % , f1 score: {:.3f}".format(accuracy * 100, f1))
```

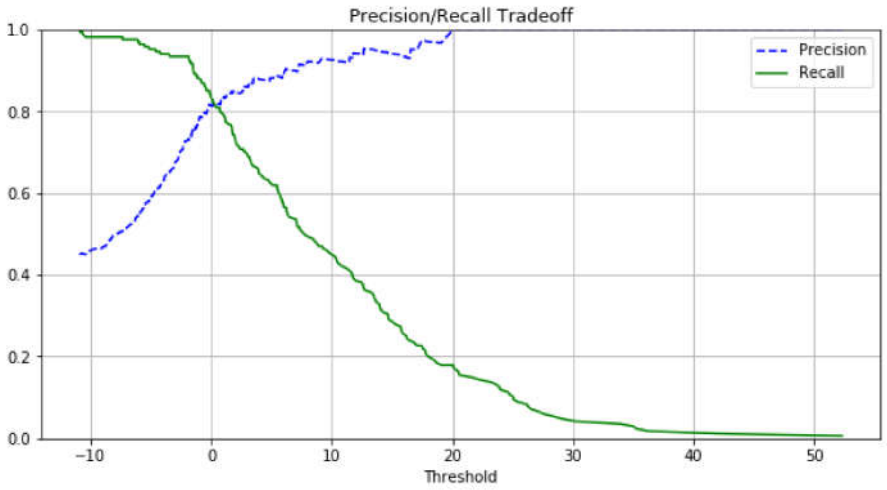
```
accuracy: 89.474 % , f1 score: 0.870
```

In [13]:
```
# let's get the scores for some instance
y_scores = sgd_clf.decision_function([x_train[0]])
# for SGDClssifier, the default threshold is 0, anything below is calssified as negative and else positive
print(y_scores)
# let's see the effect of the threshold on the training data
y_scores = cross_val_predict(sgd_clf, x_train, y_train, cv=k_folds, method="decision_function")
```

```
[-3.3631803]
```

In [14]:
```
# let's plot and see the effect
precisions, recalls, thresholds = precision_recall_curve(y_train, y_scores)
fig = plt.figure(figsize=(10,5))
ax = fig.add_subplot(1,1,1)
ax.plot(thresholds, precisions[:-1], "b--", label="Precision")
ax.plot(thresholds, recalls[:-1], "g-", label="Recall")
ax.set_xlabel("Threshold")
ax.set_title("Precision/Recall Tradeoff")
ax.legend()
ax.grid()
ax.set_ylim([0, 1])
```

Out[14]: (0, 1)



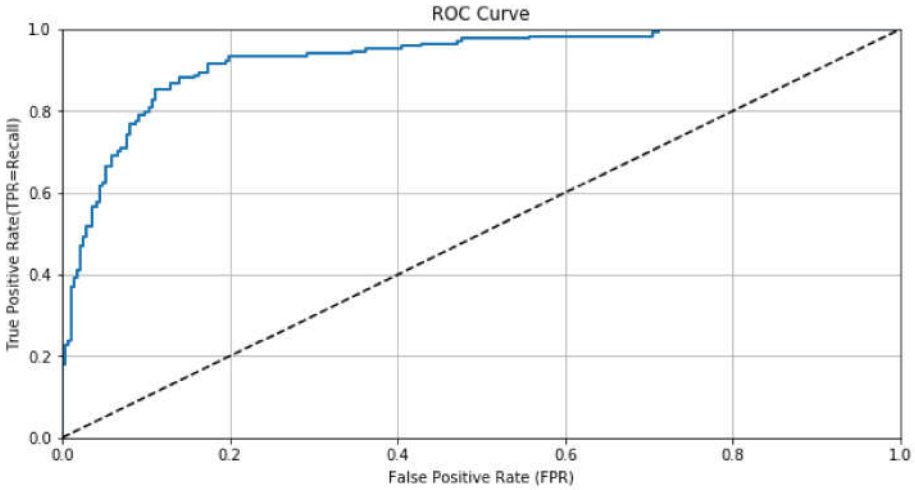
The ROC Curve

The *receiver operating characteristic* (ROC) curve is another common tool used with binary classifiers. It is very similar to the precision/recall curve, but uses the True Positive Rate (TPR, another name for Recall) vs. the False Positive Rate (FPR, the ratio of negtaive instances that are incorrectly classified as positive) we saw in the beginning of the tutorial.

- Note: The FPR is equal to $1 - TNR$ (true negative rate, or, *specifity*)
- The ROC curve plots *sensitivity* (recall) vs. 1 - *specifity*.
- **TPR/FPR Tradeoff**: The higher the TPR (recall) the more *false positive* (FPR) the classifier produces.
- To compare classifiers - measure the *area under the curve* (AUC). A perfect classifier will have a ROC AUC equal to 1, whereas a purely random one will have a 0.5 AUC.
- **Precision/Recall curve vs ROC curve**: if the positive class is rare, or you care more about the false positive than the false negative use **Precision/Recall**. Otherwise, use the ROC curve.

In [15]:
```
# the ROC curve
fpr, tpr, thresholds = roc_curve(y_train, y_scores)
precisions, recalls, thresholds = precision_recall_curve(y_train, y_scores)
fig = plt.figure(figsize=(10,5))
ax = fig.add_subplot(1,1,1)
ax.plot(fpr, tpr, linewidth=2)
ax.plot([0,1], [0,1], 'k--') # random classifier
ax.set_xlabel("False Positive Rate (FPR)")
ax.set_ylabel("True Positive Rate(TPR=Recall)")
ax.set_title("ROC Curve")
ax.grid()
ax.set_xlim([0, 1])
ax.set_ylim([0, 1])

roc_auc = roc_auc_score(y_train, y_scores)
print("ROC AUC Score: {:.3f}".format(roc_auc))
```

```
ROC AUC Score: 0.927
```



Credits

- Icons from Icon8.com (https://icons8.com/) - https://icons8.com (https://icons8.com)
- Datasets from Kaggle (https://www.kaggle.com/) - https://www.kaggle.com/ (https://www.kaggle.com/)
- Examples and code snippets were taken from "Hands-On Machine Learning with Scikit-Learn and TensorFlow" (http://shop.oreilly.com/product/0636920052289.do)