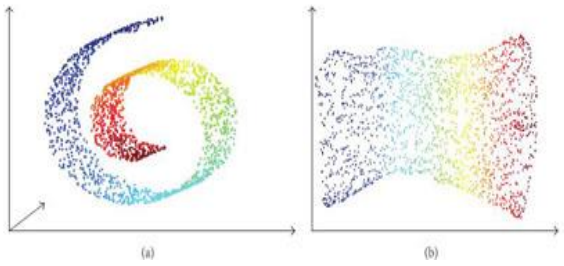


Dimensionality Reduction

MACHINE LEARNING – SERIES 1

This document explains Dimensionality Reduction. We will learn why it is important, theory and techniques used to achieve it

Viraj Vaitha

Contents

The Challenge – Curse of Dimensionality

Dimensionality reduction is an important topic for many reasons:

- Problems can involve thousands or millions of features
- Causes training to be extremely slow
- Difficult to find a good solution
- Great for Data Visualisations to understand your problem statement

This is known as the curve of dimensionality. Dimensionality reduction will usually increase the speed of your training, but often not the performance of the model (It can improve the performance, it just depends).

Reducing dimensionality causes loss of some information. While we are increasing the speed of training, it could make your system perform badly. This is not a one stop solution. There are cases where comprehending information in lower dimensions can of course simplify and improve your model

What is a Dimensional Space?

We are very used to our three-dimensional space, we have height, width and depth. The diagram below shows representations of higher dimensional spaces:

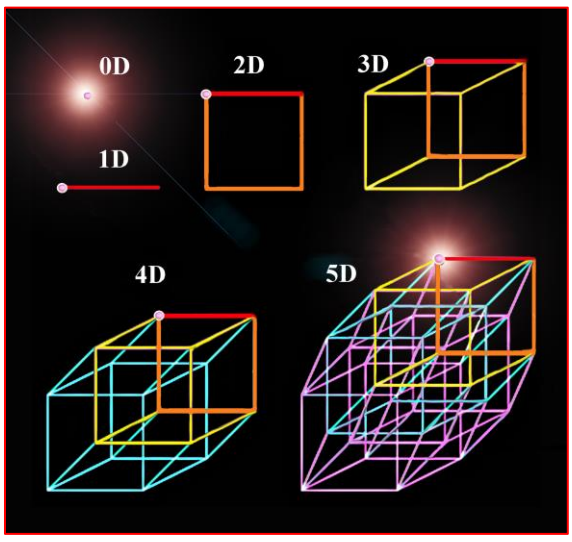


Figure 1:Dimensional Spaces

The behaviour in high dimensional space varies significantly. If you select a random point in a 1x1 square, you have less than 0.4% chance of being located less than 0.001 from a border, whereas in a 10,000-dimensional unit hypercube this probability is greater than 99.9999%. Most points lie near the border.

Some theory below:

If you select two random points in a 3d square the average distance is 0.66 whereas in a 1,000,000 the distance is on average 408.25. This means that higher dimensional data sets are at risk of being very sparse. Most training instances are likely to be far away from each other. So, the more features, the more permutations we have which increases the likely hood that the scenario or conditions will be different.

Let's try get that into English:

My understanding from this information from the extract above from the book, Hands on Scikit Learn with Tensorflow is that if we have many dimensions (feature space), the model is like a human overthinking problem. If someone said what's the price of that Ferrari, and you gave 10000 features, it may overthink the problem. In addition it might not have many previous examples to learn from where the combinations of features was similar to other models it has seen before. There is more room for small errors that accumulate in your prediction as oppose to improving it.

All you really needed to know was it is a Ferrari, the year it was made and a few other important features. Not details like the size of the compartment or colour of the pedals.

Whereas if you took a more common car that has a variety of models, maybe a higher dimensional space would better help solve the problem. Sometimes in life we don't overthink, our "guesstimation" is good enough.

The more dimensions the training set has, the greater the risk of overfitting it.

Why Increasing Training set doesn't solve the problem

In theory, increasing the number of training examples could solve the problem.

The issue is that the number of training instances needed increases exponentially with the number of dimensions.

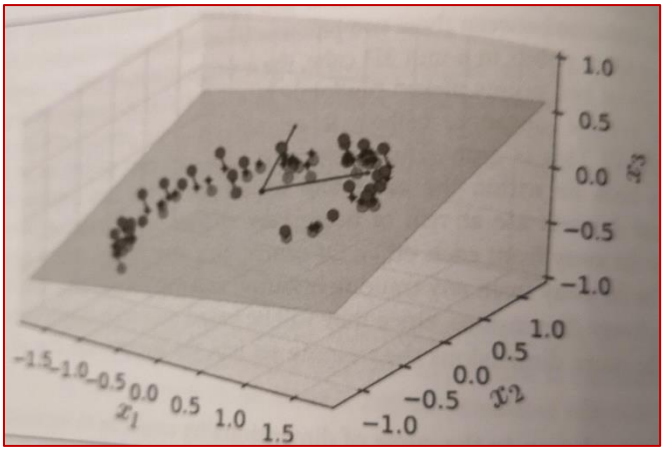
Approaches for Dimensionality Reduction

Projection and Manifold Learning are two approaches. These are approaches, as opposed to algorithms which is discussed later.

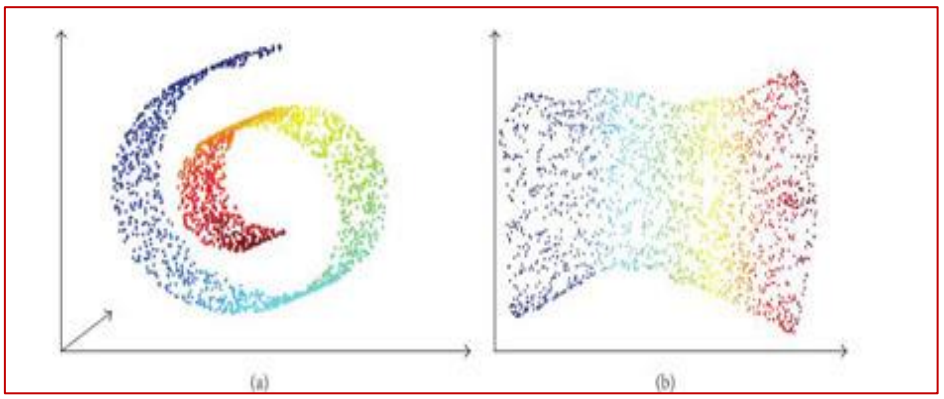
Projection

Below in image x.x, shows data plotted in a 3-dimensional space. We can project this into a 2D space. The reason projection is possible is that training instances tend not to be spread out uniformly. Most features are constant while others are highly correlated.

As you can see the data points lie close to a hypothetical 2D plane. Now we project every instance perpendicularly onto this subspace. We have reduced the datasets dimensionality.



In reality, the datapoints may not lie nicely next to this 2D plane. The points may twist and turn. This is known as the famous **Swiss roll.** This can be seen below:



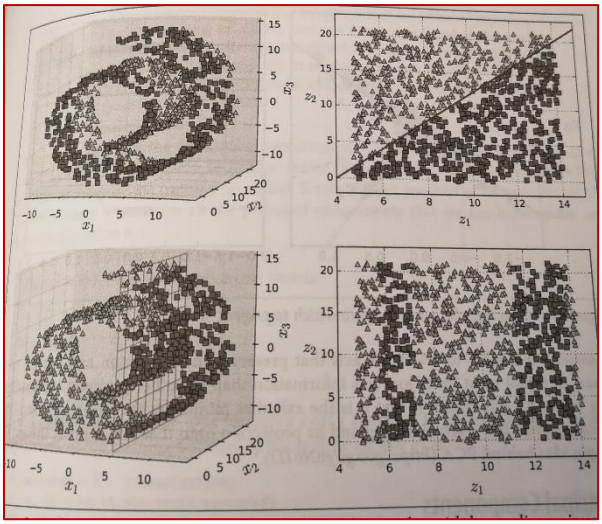
If we took the projection approach, this would squash different layers of the roll together, losing the importance of their position in the subspace.

What you really want to do is **unroll the swiss roll to obtain the dataset** (b) as seen above.

Manifold Learning

This swiss roll is an example of 2D Manifold. A d-dimensional manifold is a part of an n-dimensional space (where d <n) that can resemble a d-dimensional hyperplane. Sounds complication, but in our case it for swiss roll d =3, n =2.

Manifold assumption also known as manifold hypothesis is used in this technique. We are assuming a lower dimensional space has similar properties to that of a higher dimension. Most real world high dimensional datasets lie close to a much lower dimensional manifold.



We are also assuming that the solution will be simpler at a lower dimensional space. As we can see above this isn't always the case.

PCA

Principle Component Analysis (PCA) is a popular dimensionality reduction algorithm.

- Identifies closest hyperplane to the data.
- Projects data onto it

PCA must first select the right hyperplane. Ideally, preserving the most variance would produce great results. Selecting the axis that preserves the maximum variance means finding the axis which contains the highest variance of values. One other technique is to find the axis that minimizes the square distance between the original dataset and its projection onto that axis.

Principle Components

PCA finds the axis that preserves the most variance. Then it will look orthogonal to the axis, to find the next axis which preserves the most data. If we are in a higher dimension, we will check the third axis, fourth axis etc..

The unit vector defining the i^{th} axis is the i^{th} principle component (PC1). They are ranked in order of variance preserved. So the one with the most variance preserved will be the 1^{st} PC and so on.

The image below shows that the straight line, c1, is the first PC as it preserves the most variance.

Whereas the dashed line, c2 is the second PC.

While the direction of the principle components is meaningful, the plane will generally remain the same.

[Insert Pic]

How it's done in Python -Singular Value Decomposition

This is a standard matrix factorization.
It is used to decompose the training set matrix X into the dot product of the three matrices $U.\sum.V^T$
Essentially, the matrix v contains all the principles components and the top two are extracted.

[Insert Pic]

[Insert code]

[Research area a little bit and add knowledge]

This is an answer from online () that explained CV clearly.

when we say 'a model' we refer to a particular method for describing how some input data relates to what we are trying to predict. We don't generally refer to particular instances of that method as different models. So you might say 'I have a linear regression model' but you wouldn't call two different sets of the trained coefficients different models. At least not in the context of model selection.

So, when you do K-fold cross validation, you are testing how well your model can get trained by some data and then predict data it hasn't seen. We use cross validation for this because if you train using all the data you have, you have none left for testing. You could do this once, say by using 80% of the data to train and 20% to test, but what if the 20% you happened to pick to test happens to contain a bunch of points that are particularly easy (or particularly hard) to predict? We will not have come up with the best estimate possible of the model's ability to learn and predict.

We want to use all of the data. So to continue the above example of an 80/20 split, we would do 5-fold cross validation by training the model 5 times on 80% of the data and testing on 20%. We ensure that each data point ends up in the 20% test set exactly once. We've therefore used every data point we have to contribute to an understanding of how well our model performs the task of learning from some data and predicting some new data.

But the purpose of cross-validation is not to come up with our final model. We don't use these 5 instances of our trained model to do any real prediction. For that we want to use all the data we have to come up with the best model possible. The purpose of cross-validation is model checking, not model building.

Now, say we have two models, say a linear regression model and a neural network. How can we say which model is better? We can do K-fold cross-validation and see which one proves better at predicting the test set points. But once we have used cross-validation to select the better performing model, we train that model (whether it be the linear regression or the neural

network) on all the data. We don't use the actual model instances we trained during cross-validation for our final predictive model.

MNIST Dataset

MNIST dataset is a collection of handwritten digits (greyscale) images. It is very common for new Machine Learning developers to experiment with MNIST, Titanic, Automobile dataset and more.

The aim is to predict the handwritten digit from an image. The ability to predict the digit from a scanned image could lead to many innovative technologies. Let's say extracting text from files (pdf files). Essentially, we want computers to be able process and gain information from images. Remembering the video footage is simply "Frames per second", a collection of images assembled together. As humans our eyesight collects the footage while our brain processes and gains meaning from the from the visuals we see.

This document will focus on handling/plotting images and building a binary classifier. A binary classifier's in this case would be to classifier whether the image is a 5, or not a 5. (You can do it for any number). We will then explore common metrics used to assess the performance of classifiers. This includes precision, recall, accuracy, F1 score, AUC and the confusion matrix.

Popular datasets can be accessed through *scikit-learn*.

```
from sklearn.datasets import load_digits

MNIST = load_digits()  #Here we loaded the dataset into the variable name MNIST
```

Now we have imported the dataset. The datasets are usually packaged quite well to make it easier to work with through Scikit-Learn. The data is stored as an array, an efficient way to sore data for processing images since all the values are numeric.

'data' contains an array of numbers that represents the intensity of pixels which build up an image. Therefore, each image will be X by X Matrix of values. The 'target' is the label of each image, *i.e array of pixels that represent 5, and the label is 5*. For each image we have an associated label. This will become a lot clearer as you work with more arrays.

```
In [6]: MNIST #data array contains the features
           #target contains the associated labels

Out[6]: {'data': array([[ 0.,  0.,  5., ...,  0.,  0.,  0.],
                [ 0.,  0.,  0., ..., 10.,  0.,  0.],
                [ 0.,  0.,  0., ..., 16.,  9.,  0.],
                ...,
                [ 0.,  0.,  1., ...,  6.,  0.,  0.],
                [ 0.,  0.,  2., ..., 12.,  0.,  0.],
                [ 0.,  0., 10., ..., 12.,  1.,  0.]]),
         'target': array([0, 1, 2, ..., 8, 9, 8]),
         'target_names': array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]),
         'images': array([[[ 0.,  0.,  5., ...,  1.,  0.,  0.],
                [ 0.,  0., 13., ..., 15.,  5.,  0.],
                [ 0.,  3., 15., ..., 11.,  8.,  0.],
                ...,
                [ 0.,  4., 11., ..., 12.,  7.,  0.],
                [ 0.,  2., 14., ..., 12.,  0.,  0.],
                [ 0.,  0.,  6., ...,  0.,  0.,  0.]],

               [[ 0.,  0.,  0., ...,  5.,  0.,  0.],
                [ 0.,  0.,  0., ...,  9.,  0.,  0.],
                [ 0.,  0.,  3., ...,  6.,  0.,  0.],
                ...,
                [ 0.,  0.,  1., ...,  6.,  0.,  0.],
                [ 0.,  0.,  1., ...,  6.,  0.,  0.],
                [ 0.,  0.,  0., ..., 10.,  0.,  0.]],
```

Let's assign the data to the variable X, and labels to variable y.

```
X, y = MNIST["data"], MNIST["target"]
```

```
X_train, X_test, y_train, y_test = X[:1079], X[1079:], y[:1079], y[1079:]
```

Handling Images

a) Plotting an Image (Python)

Images can be plotted through matplotlib. The general process involves importing modules, reshaping images and plotting the image.

The scenario of a grayscale image is simple. Each feature (column of data) is usually the **intensity of the pixel,** from **0 (white) to 255(black**). Imagine a grid of light bulbs where each number in the grid represents the intensity of the bulb. Looking at the grid would now show a number. (below a grid of coloured bulbs to show a pattern.

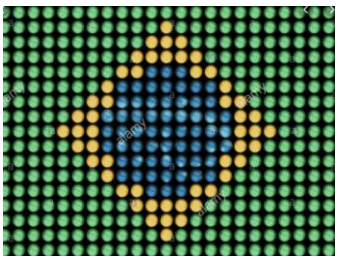


Figure 2:Image to describe how pixels create an image

The MNIST dataset that was imported from scikit learn has the shape (1079, **64**). The first value is the number of row's (number of images), whereas the second is the number of **columns.** We can take the root of this value to help us shape the data to display the image. In this case it is 8x8 image.

Resizing an image is a common task, for now all images should have the same dimensions for a certain machine learning problem. The memory limitation can be a problem with images. By scaling down or cropping our images we can save a lot of memory. Curious to see how model accuracy changes with higher or lower resolution.

Reshaping an image is useful. For example, in MNIST we have 70000 images with 784 features. However, to view this image, we should **extract one** image of the database **(one row), giving (1,784)** and **reshape it to say 28x28**. This will enable us to see the image.

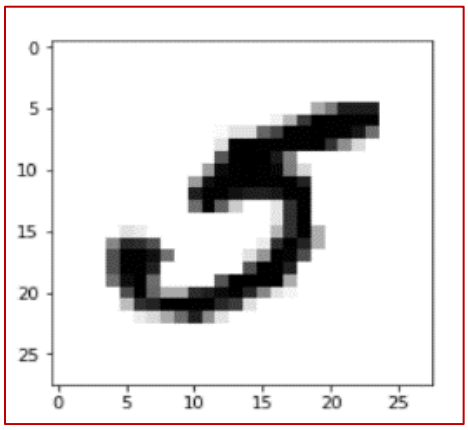


Figure 3:Plotted Images through Matplotlib

Plotting Images

Step 1: Import Relevant Modules

```
%matplotlib inline

Import matplotlib

Import matplotlib.pyplot as plt
```

Step 2: Select a random image from the test set

```
Image = X_test[24] #Number could be anything, selected 24th index position, from the dataset
```

Step 2: Resize or Reshape the image

```
Image.reshape(8,8)  #Shaping as 28x28, grid of light bulbs
```

Step 3: Plot the image

```
plt.imshow(image, cmap= matplotlib.cm.binary, interpolation ="nearest")

plt.axis("off")
plt.show()
```

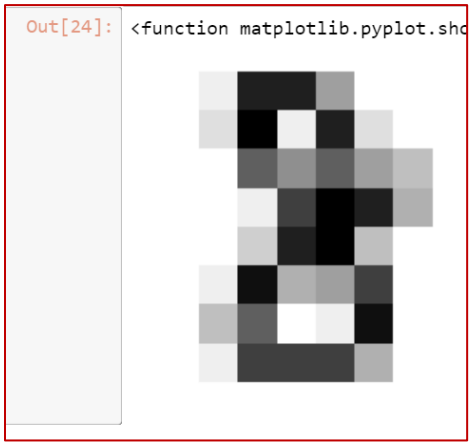


Figure 4:Lower Resolution Image plotted

Handling Images

b) Binary Classifier for Images

For a binary classifier we require **labels, 0 and 1 (True/False)**. In this case we can use our existing labels and convert them to **true and false**.

We have numbers from 0-9. Instead of trying to classifier each number, for now we will try to classify if a number is a 5 or not 5. This is an example of **Binary Classification.** Technique used to classify between two classes.

We should initiate a **random state** since the **SGDClassifier** operates on randomness. We want **reproducibility** in our results. Otherwise our model will behave differently each time (More on the actual SGDClassifier in future booklets, for now you just need to know it is an algorithm that can be used for binary classification)

Finally, **fit** our data to the new labels and **predict**

Step 1: Convert the value columns to two labels. Existing Labels (y values)

```
#Boolean Technique, each value in that column is set to == 5, resulting in true or false labels.

y_train_8 = (y_train == 5) # == will equate each row in the array to 5. If it is TRUE its ill produce
                           # TRUE, otherwise FALSE

y_test_8 = (y_test == 5)   #Same Operation for the test set
```

Step 2: Fit the model (Fitting the model means training the model based on the training data)

```
From sklearn.linear_model import SGDClassifier
Sgd_clf = SGDClassifier(random_state =42) #Initiated the model
Sgd_clf.fit(X_train, y_train_5)           #Fit the model based on this data and their labels
```

Step 3: Predict on a single digit

```
#Test on a digit
Image = X_test[24]
Sgd_clf.predict([image])                  #Did it predict it correctly?
```

Handling Images

c) Cross Validation Implementation

Cross Validation using Accuracy

Evaluating Classifiers can be difficult. Sometimes we may want more control over cross validation. The method describes below is as if we are manually doing the cross validation. (I will still use simplified way seen on the next page)

We will define:

- Number of splits (How many times we want to test our model on different "sets or splits" of data).
- Random State (This ensure we can have reproducibility in our results due to our splits)

Below is a manual implementation of cross validation, **a function already exists to execute Cross Validation in a few lines of code. This can be seen below the proceeding example.**

Step 1: Import Relevant Modules

```
from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone
```

Step 2: Prepare to split the data into folds and set random state

```
skfolds = StratifiedKFold(n_splits=3,random_state=42)
```

Step 3: Loop through the dataset

```
for train_index, test_index in skfolds.split(X_train, y_train_8):
```

```
    clone_clf = clone(sgd_clf)
    X_train_folds = X_train[train_index]
    y_train_folds = y_train_5[train_index]
    X_test_fold= X_train[test_index]
    y_test_fold = y_train_5[test_index]
    clone_clf.fit(X_train_folds, y_train_folds)
```

Step 4: Predict

```
y_pred = clone_clf.predict(X_test_fold)
n_correct = sum(y_pred == y_test_fold)    print(n_correct/len(y_pred)
```

Alternatively:

```
from sklearn.model_selection import cross_val_score
cross_val_score(sgd_clf, X_train,y_train_5, cv=3, scoring="accuracy")
```

Evaluating your Model

a) Baseline Model

https://stackoverflow.com/questions/15233632/baseestimator-in-sklearn-base-python

`BaseEstimator` provides among other things a default implementation for the `get_params` and `set_params` methods, see [the source code]. This is useful to make the model grid search-able with `GridSearchCV` for automated parameters tuning and behave well with others when combined in a `Pipeline`.

We must always create a baseline model, otherwise we have nothing to compare to.

In this case our baseline model can be always NOT 5.

```
from sklearn.base import BaseEstimator
class Never5Classifier(BaseEstimator):
    def fit(self, X, y=None):
        pass
    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)
never_5_clf = Never5Classifier()
cross_val_score(never_5_clf,X_train,y_train_5, cv=3, scoring ="accuracy")
```

This produces an average of over 90%, it shows that accuracy can be deceiving for classifiers. As there are over 90% non 5's in the dataset by predicting not 5 we can get over 90%. At least the model we will create beat this benchmark.

Tip: For those of you that are interested, the Base Estimators purpose is that you could create a custom estimator (Never5 in our case) and quickly fit it to the data, get predictions, use it in CV and other sci kit learn modules. Most importantly use it in your sci kit learn pipelines which we learn about when we do an end to end machine learning project. (I am less confident on this area so please refer to documentation too)

Evaluating Your Model

a) Confusion Matrix

Example

We could create a model that labelled all values as False, (Not 5). This would yield a high accuracy as only 10% of the values were 5's. The confusion matrix will help us understand exactly how our classifier is performing.

Obtaining Confusions Matrix

Step 1: Import module

```
From sklearn.metrics import confusion matrix
```

Step 2: Run Cross Validation Predict (NOT CV score)

```
#We require the predictions themselves to determine the confusion matrix. You will use cv score and cv predict. When to use which will become intuitive over time.

y_train_pred = cross_val_predict(sgd, X_train, y_train_5, cv=3)
```

Step 3: Pass the arguments, labels and predictions into the confusion matrix

Confusion_matrix(y_train_5, y_train_pred)

Understanding the Matrix

True Negatives – Correctly classified as non 5's	False positives – Wrongly classified as 5's
False Negatives- Wrongly classified as non 5's	True Positives – Correctly classified as 5's

```
In [42]: confusion_matrix(y_train_5, y_train_pred)
Out[42]: array([[53875,   704],
                [ 1301,  4120]])
```

Precision and Recall

Precision and Recall are two important performance measures to evaluate our model.

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

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

Import Relevant Module

```
from sklearn.metrics import precision_score, recall_score


precision_score(y_train_5, y_train_pred)     # Arguments are actual values and predictions
recall_score(y_train_5, y_train_pred)
```

```
In [24]: precision_score(y_train_5, y_train_pred)
Out[24]: 0.8540630182421227

In [25]: recall_score(y_train_5, y_train_pred)
Out[25]: 0.7600073787124146
```

Let's understand what this means. This concept can be a tough one to wrap your mind around as there are a few permutations.

To summarize with our example:

True Positive (TP) is whether the value was correctly classified as a 5. Whereas a False Positive is when a class was incorrectly classified as a 5.

True Negative (TN) is whether a value was correctly classified as a non 5. Whereas a False negative is when a class was incorrectly classified as a non 5.

Breakdown to further articulate this concept:

The True/False part describes whether the class was correctly or incorrectly classified.

The Positive/Negative part describes whether we are talking about the 1 (is the class) or 0 (not the class). In our case whether we are discussing in terms of a 5 or not a 5.

Precision tells us, how many of the positive class was correctly classified. In this case, 5 is our positive class therefore high precision shows we are confident in classify a 5's.

100% precision would mean, whenever we classify it is a 5, it will be a 5. The example in the blue box describes why this performance measure is so important.

Recall tells us, are we falsely classifying the negative class as a positive class. In this case, a non 5 is our negative class therefore high recall shows we are not mistaking non 5's as 5's.

Example

Let's say we want to detect Safe Videos and filter out bad videos for children.

High precision means bad videos are not mistakenly classified as safe.

High Recall means safe videos are not falsely classified bad videos

So we can see "Recall" does not matter much to us since we would rather have safe videos classed as bad, as opposed to bad videos classified as safe.

Essentially, we compromise Recall to increase Precision, this is the Precision /Recall Trade off.

Finally, we can break this down into two statements.

Either we want to be confident when we predict the positive class (bad videos or is 5's), at the stake of labelling some negative classes (safe videos or non 5's) as positive classes (bad videos or 5's)

Or

Confident in predicting the negative class (safe videos or non 5's), at the stake of labelling some positive classes (bad videos or 5's) as negative classes (safe or non 5's)

F1 Score

The F1 Score incorporates **both** Precision and Recall into its calculation.

This is a **harmonic mean of precision and recall**. A regular mean would treat all values equally whereas harmonic mean gives much more weight to low values.

F1 score favours classifiers that have a similar precision and recall. In cases where precision or recall is more important, we should not use the F1 Score. Accuracy and F1 Score can be used to give a good idea of the performance of a classifier (that does not require high precision/recall)

Example

Let's say we want to detect Safe Videos and filter out bad videos for children. If precision is high, this means we are not labelling bad videos as good which is essential. Recall is low meaning that there may be some good videos that are wrongly classified as bad. But at least the kids didn't see any bad videos. #Repeated #EmphasizingP/RTradeoff

Step 1: Import Relevant Module

```
from sklearn.metrics import f1_score
```

Step 2: Use Module

```
#y_train_5 = labels of trainset. y_train_pred = predictions we recieved.

f1_score(y_train_5, y_train_pred))
```

```
In [26]: from sklearn.metrics import f1_score
         f1_score(y_train_5, y_train_pred)

Out[26]: 0.8042947779404588
```

Precision/Recall Graph

Viewing the data visually will help find the threshold that best suits your project. We must utilise the decision_function argument in order to plot this graph.

CV using Decision Function

```
#This will enable the graph to be plotted for each threshold
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv =3, method = "decision_function")
```

Import Relevant Module

```
#This module is used to obtain precision and recall for all thresholds.
from sklearn.metrics import precision_recall_curve
```

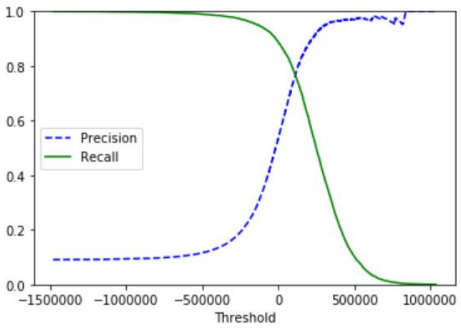
Determine values for each thresholds

```
#y_train_5 is the training labels and y_scores includes predictions for all thresholds as scores
precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

Plot the Graph

```
def plt_precision_recall_vs_threshold(precisions,recalls,thresholds):
    plt.plot(thresholds,precisions[:-1],"b--",label="Precision")
```

```
plt.plot(thresholds, recalls[:-1],"g-", label= "Recall")
plt.xlabel("Threshold")
plt.legend(loc="center left")
plt.ylim([0,1])
```



Obtaining Precision and Recall for new thresholds

Create Function

```
def PrecisionRecall(TrainingSet, Threshold):
    Threshold_Predictions = (y_scores> Threshold) #Adjusts prediction scoring for new threshold
    a = precision_score(TrainingSet,Threshold_Predictions) #Calculates Precision
    b = recall_score(TrainingSet, Threshold_Predictions) #Calculates Recall
    print ("Precision = " + str(a))
    print ("Recall = " + str(b)) #Prints values
```

Precision vs Recall Graph

I believe the best method is to plot the Precision vs Recall graph as seen below.

You have determined what precision you want, as cost of what recall. Now simply use the Precision Recall Threshold graph to determine the relevant threshold.

The ROC Curve – Great Graph for comparing different classifiers

Summary: Area under the curve = 1 is a perfect classifier. The further to the top left corner, indicates a better classifier. Quick way to compare various classifiers.

TPR – Recall (Sensitivity)

The ratio of positives labelled correctly as positive. True Positive Labels/ Total Positive Labels

FPR = 1 – TNR (Specificity)

FPR is the ratio of negative instances that are incorrectly classified as positive. TNR is the ratio of negative instances that are correctly classified as negative.

Import Relevant Modules

```
From sklearn.metrics import roc_curve

Fpr,tpr,thresholds = roc_curve(y_train_5, y_scores)
```

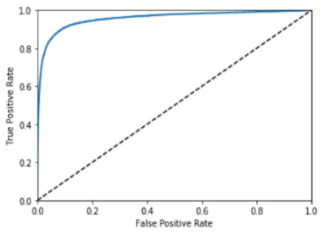
Plot Graph

```
def plot_roc_curve(fpr,tpr, label=none):
    plt.plot(fpr,tpr,linewidth = 2, label=label)
    plt.plot([0,1],[0,1], 'k - -' )
    plt.axis([0,1,0,1])
    plt.xlabel('False Positive Rate)
    plt.ylabel('True Positive Rate')
plot_roc_curve(fpr,tpr)
plt.show
```

We see there is a tradeoff. As TPR increases the FPR starts to increase. This indicates that the higher TPR, the higher FPR will be. Knowing the FP's that initially form would be useful.

This can be used to compare classifiers. The more far to the top left the better the classifier. The AUC would be a numeric value (area under the curve)

Recall



1-Specificity

AUC Score

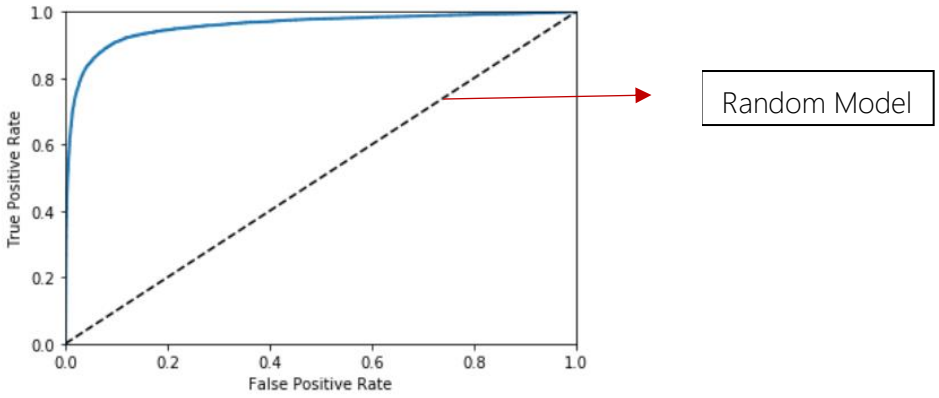
Good classifiers stay far away from the middle line of the ROC curve.
The Area under the curve = 1 is a perfect model.

Import Modules

```
From sklearn.metrics import roc_auc_score
```

Apply Module

```
Roc_auc_score(y_train_5,y_scores)
```



IF:

Positive Class is rare

OR

False Positives more Important than False Negatives (Bad Videos being mistaken as safe content example)

Use PR Curve.

Else:

ROC curve.

ROC can be misleading if the rules above not followed:

Our model could look great. Especially because there are a few labels (5's) compared to other numbers. This ratio explains ROC well. Having a few 5's and many other numbers (1,2,3,4 ect..) makes it seem that the job is great. Always check if this is the case before using ROC.

Comparing Two Models on one ROC Graph

Import Relevant Modules

```
from sklearn.ensemble import RandomForestClassifier
```

Fit Random Forest Classifier to the Train set (Cross Validation)

```
forest_clf = RandomForestClassifier(random_state=42)
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3, method ='predict_proba')
```

Obtain Probabilities as scores for random forest

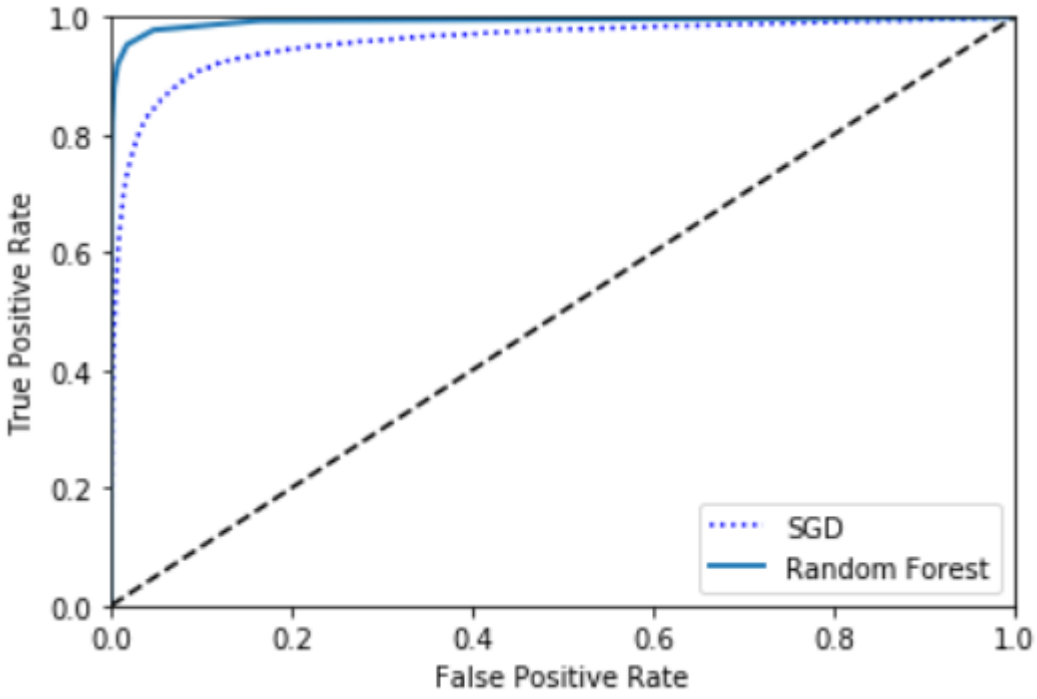
```
y_scores_forest= y_probas_forest[:,1]
```

Obtain Values to be plotted

```
fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5, y_scores_forest)
```

Plotting the graph

```
plt.plot(fpr,tpr, 'b:', label = "SGD") #Manually plot one curve
plot_roc_curve(fpr_forest,tpr_forest,"Random Forest") #Function created earlier. Includes all the other properties of the graph
plt.legend(loc="lower right")
plt.show()
```



Binary Classification Methodology

Determine your two classes and ensure you have binary labels.

Create your Machine Learning Classifier

Ask yourself, is Precision or Recall more important for your model?

Yes, it is:

Plot P & R Graph, and determine the optimum ratio

Plot P&R with Threshold values to determine the Optimum Threshold for your use your classifier.

Repeat for several models

Determine the AUC or Plot ROC Graph to help finalise the best model for your use case.

No, it isn't:

Determine what type of model would be the best, use f1 score/accuracy to compare initially and tweak your precision and recall to get the type of model you want (for example, maybe you want a well balance model)

From	Import	Example	Notes
sklearn.linear_model	SGDClassifier	• Sgd_clf= SGDClassifier(random_state =42) • Sgd_clf.fit(X_train, y_train_5) • Sgd_clf.predict([Some_Digit])	This is a linear model. Random_state for reproducibility.(SGD operates on randomness) .fit arguments are training_set, labels .predict arguments is a matrix/vector [ ]
Sklearn.model_selection	StartifiedKfold	• skfolds = StratifiedKFold(n_splits=3,random_state=42) for train_index, test_index in skfolds.split(X_train, y_train_5): • See code in document to finish off CV with Stratified sampling	Shuffles the dataset in a way that makes each set contain a diverse range of possible values in each training set.
	cross_val_score	cross_val_score(sgd_clf, X_train,y_train_5, cv=3, scoring="accuracy")	Scoring has many possible arguments such as precision. Search online to see more. Arguments is the model, train, testset and cv folds.
	cross_val_predict	y_train_pred = cross_val_predict(sgd, X_train, y_train_5, cv=3) additional argument is method = “decision_function”.	Same as above. However this will return the predictions as opposed to the scoring. Decision_function will produce predictions that are a score in order calculate certain metrics while altering the threshold.

sklearn.metrics	precision_score,recall_score	precision_score(y_train_5, y_train_pred) recall_score(y_train_5, y_train_pred)	Arguments to pass are the training labels and the predictions
	f1_score	f1_score(y_train_5, y_train_pred)	
	precision_recall_curve	precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores) Graph def plt_precision_recall_vs_threshold(precisions,recalls,thresholds): plt.plot(thresholds,precisions[:-1],"b--",label="Precision") plt.plot(thresholds, recalls[:-1],"g-", label= "Recall") plt.xlabel("Threshold") plt.legend(loc="center left") plt.ylim([0,1]) plt_precision_recall_vs_threshold(precisions,recalls,thresholds) plt.show()	When you want to assess the impact of the threshold use this function. The first section of code will obtain the parameters you require for you. (p, r and t) The decision_function of cross_val_predict must be used to produce y_scores to use this.
	roc_curve	From sklearn.metrics import roc_curve Fpr,tpr,thresholds = roc_curve(y_train_5, y_scores) Plotting the Graph def plot_roc_curve(fpr,tpr, label=none):	

		plt.plot(fpr,tpr,linewidth = 2, label=label) plt.plot([0,1],[0,1], 'k - -') plt.axis([0,1,0,1]) plt.xlabel('False Positive Rate) plt.ylabel('True Positive Rate') plot_roc_curve(fpr,tpr) plt.show	
	Roc_auc_score	Roc_auc_score(y_train_5, y_scores)	Arguments to pass are the labels and score predictions from decision_function