**Model Evaluation and Selection:**

**Classification:**

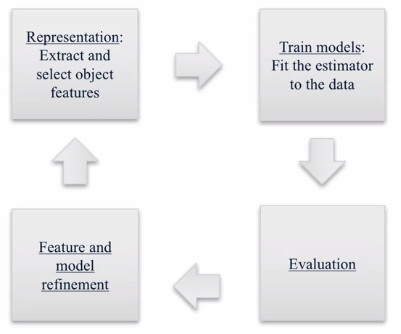
We have previously used accuracy as our metric to evaluate our classification models, this is the percentage of the classifiers predicted labels that match the correct or true labels.

**Regression:**

We also looked at using the R^2 method for evaluating a regression model, which gives us a value between 0-1.

**Learning Objectives:**

* Understand why accuracy only gives a partial picture of a classifier’s performance.
* Understand why the motivation and definition of important evaluation metrics in machine learning.
* Learn how to use a variety of evaluation metrics to evaluate supervised machine learning models.
* Learn about choosing the right metric for selecting between models or doing parameter tuning.



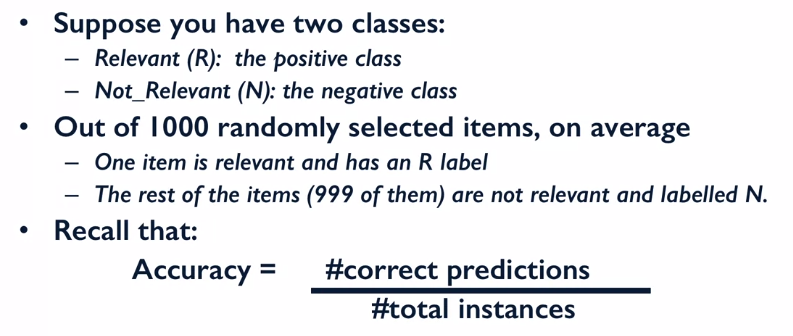
**Evaluation:**

Accuracy is widely used, but many other are possible, e.g. user satisfaction, amount of revenue, increase in patient survival rates.

* It’s important to choose an evaluation method that match the goal of your application.
* Compute your selected evaluation metric for a few different models.
* Then select the model with the best value of evaluation metric.

**Example:**

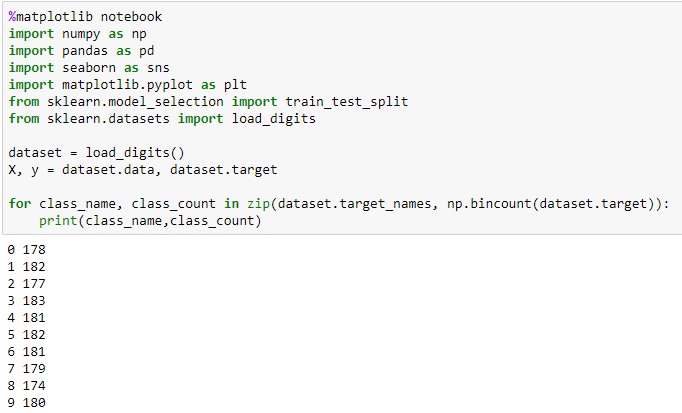
The following example shows how just using accuracy is not a good measure of model performance for some situations. E.g. credit card fraud, the majority of transactions are normal but maybe 1/10000 are not. This is an example of an **imbalanced class.** The slide below can be considered as an anti-fraud detection model:



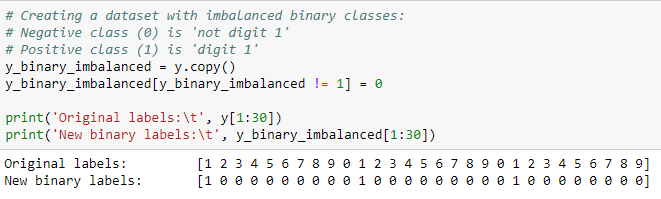
We get an accuracy of 99.9%! However, we have created a **dummy** **classifier** that didn’t look at the features and blind predicted the most frequent class (i.e. the negative N class). The 99.9% accuracy actually never predicted fraud…

**Code Example:**

We’re using a data set used to predict the value of a handwritten integer. Using NumPy’s bincount we can find the number of integers in each class.

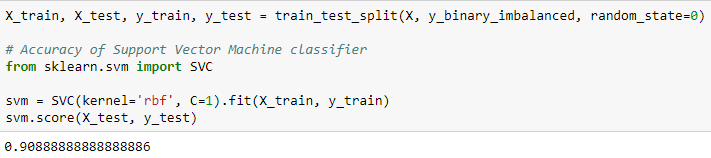


Now let’s create an **imbalanced class** of data. We do this by converting the data into either 1’s or 0’s if the data was originally a 1 it will remain a 1, otherwise it will be 0.





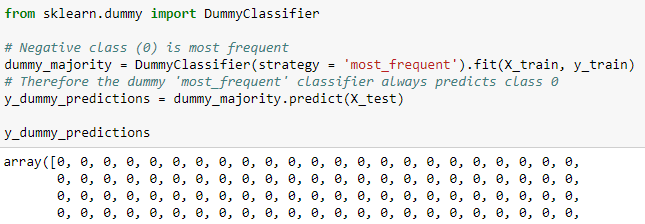
We can see that now this data set has a very large class imbalance. Let’s now train a SVM to predict between these two classes.



**Dummy Classifiers**

Wow 90.9% accuracy! However, if the model just blindly predicted that the value would be zero, we would result in an accuracy of 89.9% (1615/ (1615+182)), therefore our model is hardly better than a **dummy classifier**.

Let’s use a dummy classifier to see how well it predicts the imbalanced classes used above:





We can see that the SVM model and the dummy model predict pretty much with the same accuracy!

The **dummy classifier** provides what is known as a **null metric baseline**. This is the accuracy that is always achieved by picking the most frequent class.

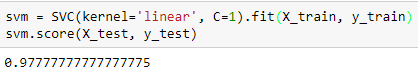
There is more than one strategy when using DummyClassifer(), they can be seen below:

* **Most\_frequent**: This is the one used in the example above.
* **Stratified**: random predictions based on training set class distribution, similar to the most frequent but it randomly applied the class prediction to the instances, so there will be 2 predictions rather than just 1.
* **Uniform**: generates predictions uniformly at random, all classes have an equal chance of being output rather than basing it on the frequency of the class labels in the data.
* **Constant:** always predicts a constant label provided by the user. Can be useful when computing **F1 score**.

**What if our Classifier has accuracy close to the null accuracy baseline?**

* Ineffective, erroneous or missing features.
* Poor choice of kernel or hyperparameter.
* Large class imbalance.

E.g. if we redo the above SVM but instead us a linear kernel we can see that the model performs well:

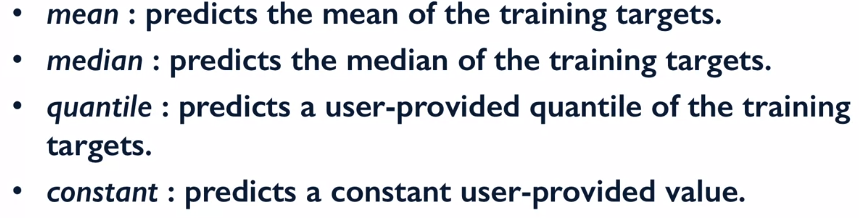


For null accuracy due to large class imbalance we should use metrics other than accuracy, instead use a metric called **Area under the curve (AUC)**, we’ll go into more details later on.

**Dummy Regressors:**

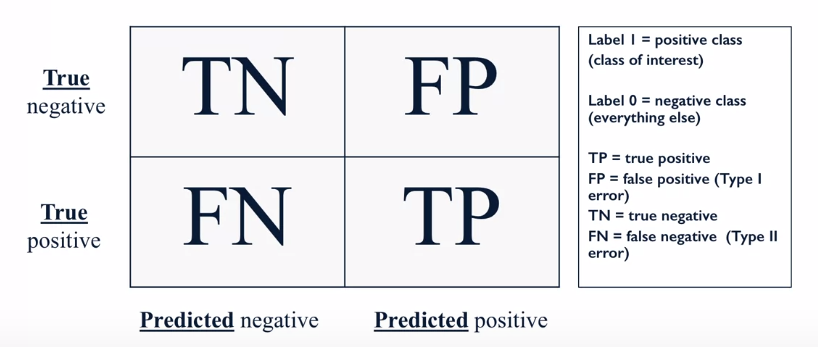
They serve an identical role for regression problems and serve as a baseline and sanity check for our models’ performance.

Since regressions output a continues value as a prediction, dummy regressors give you a choice of functions that you can apply to the training set.

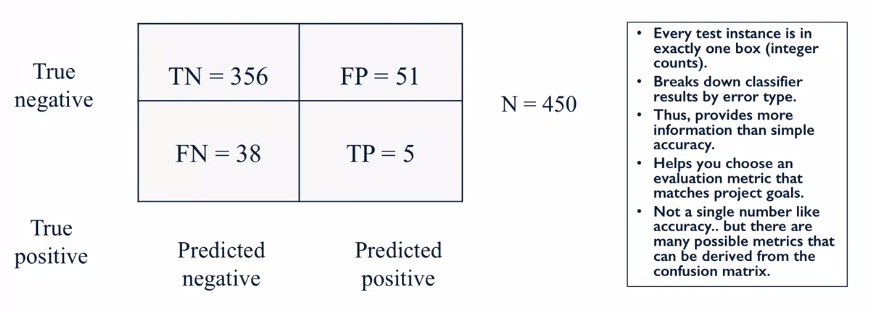


**Binary Classifier Outcomes:**

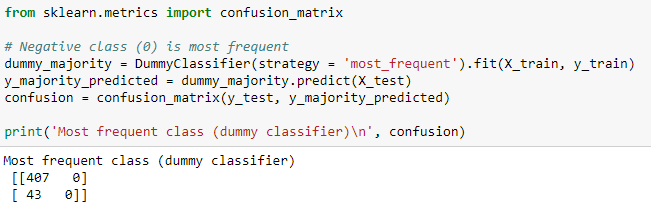
Below shows a **confusion matrix**.



For a classifiers output it can be either the first (if class prediction is negative) or last (if the class is predicted positive) columns of the matrix above. If the class is predicted negative and it is indeed negative, then it is a **True Negative TN**. If the class predicts negative and the value is actually positive, then it’s a **False Negative FP**. If the model predicts positive and the value is actually negative, then this is a **False Negative FN**. If the model predicts a positive outcome and the true value is actually a positive, then it’s a **True Positive TP**.



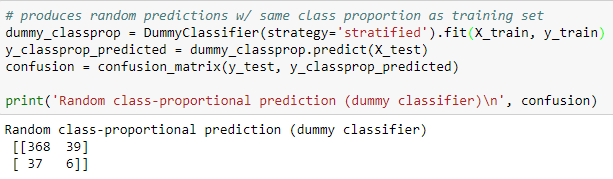
**Confusion Matrix Code Example:**



TN = 407, FN = 43. The correct model predictions lie on the left to right diagonal of the **confusion matrix** and the right to left diagonal shows the errors the model makes.

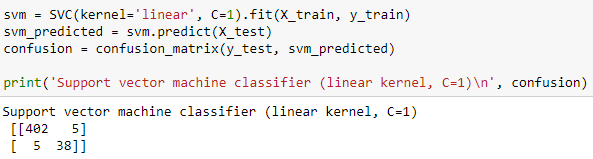
Above we can see that the model has only predicted negative values (e.g. 0), and that 43 of these predictions were wrong.

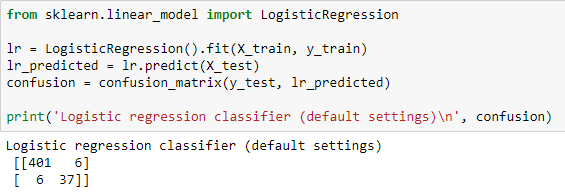
Let’s try a dummy classifier with a **stratified** strategy:

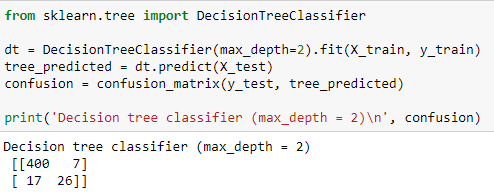


We have values in both columns of the matrix because this strategy does randomly pick negative and positive predictions (0,1 respectively) based on the distribution of classes. We can see that it miss-predicted 39 of the positive values (1’s), and 37 of the negative values (0’s).

Let’s try a few classification models to see which produces the best values:



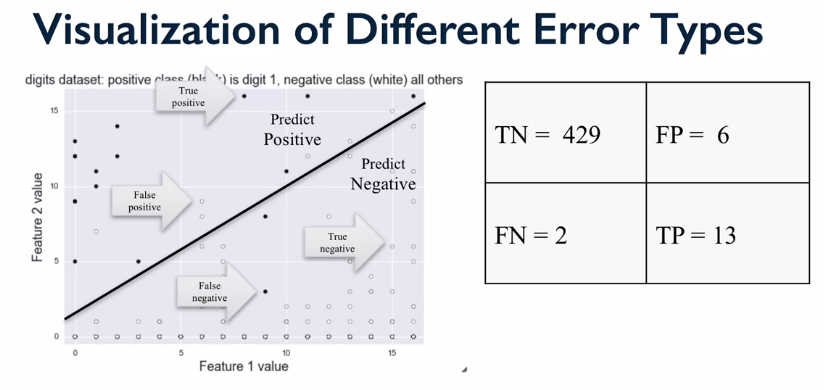




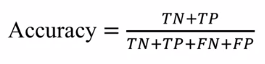
We can see that all the models used above produce a very high accuracy. However, the decision tree makes more than 2x more mistakes when classifying 0’s. This would not have been noticed if only accuracy was being measured.

**Confusion Matrices and Basic Evaluation Metrics:**

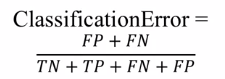
We can visualize the errors of FP and FN:



**Accuracy** is the total number of correct predictions (left to right diagonal) over the total number of predictions. This is used when the **True Positive** and **True Negatives** are more important. Accuracy **shouldn’t be used on imbalanced classes.**



**Classification Error** is the total number of incorrect predictions (right to left diagonal) over the total number of predictions:



**Recall** Its used to evaluate how well a model is at **dealing with FN**, this is often called the **True Positive Rate**. For example, this might be really important with Covid tests, as having loads of FN would result in more people getting infected.



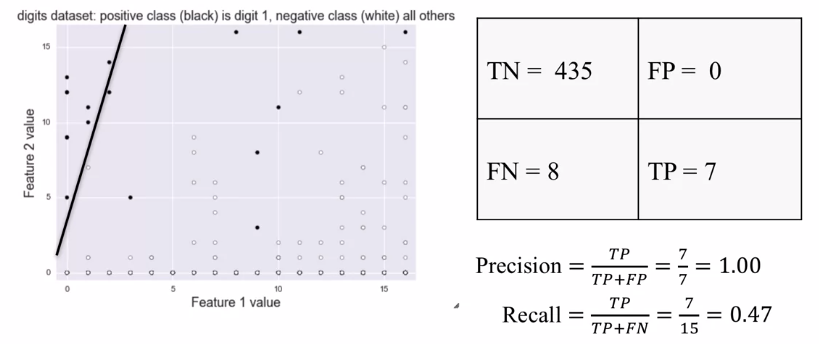
**Precision** is used to evaluate how well a model is at **dealing with FP**. This is normally used in customer facing prediction problems like YouTube recommendation systems, it does matter to much if we’re not recommended a video that we would love (FN), but it would be strange to be recommended a video that was completely not for us. Users are more likely to remember wrong instances than the right ones.



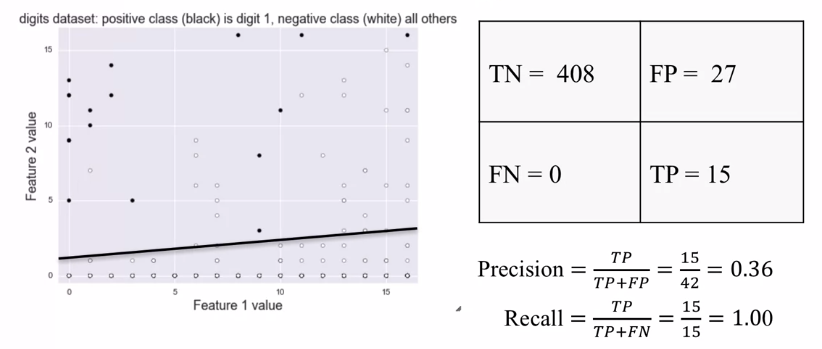
**False Positive Rate (FPR)** this is the fraction of all **negative instances** that the classifier **incorrectly** identify as **positive**.



If the we wanted to make a recommender system that users would like we would aim to have high precision and might skewed the decision boundary to do this:



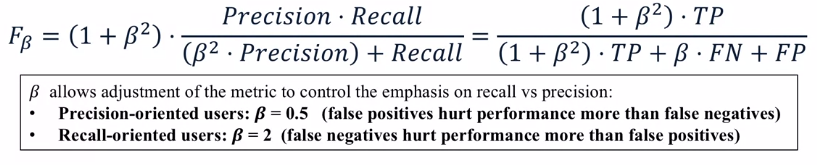
If we wanted to use machine learning to predict the outcomes of a Covid test, then we would want to have very high recall scores and would skew the decision boundary to achieve this:



**F1-score** is used to combine the recall and precision metrics into one output.

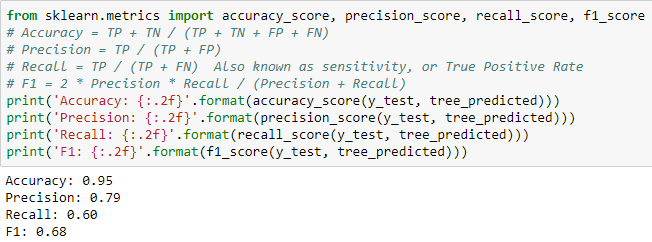


A more complex for of F1-score can be used if there is a desired weighting, we want to give to either precision or recall. Sometimes we want there to be good precision and decent recall also (and vice versa).

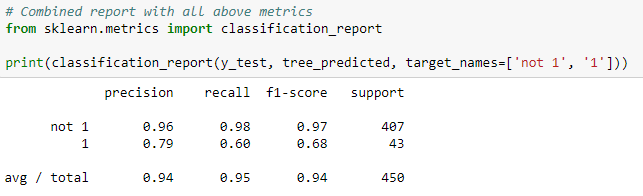


**F1 score** is used when the **False Negative** and **False Positives** are crucial.

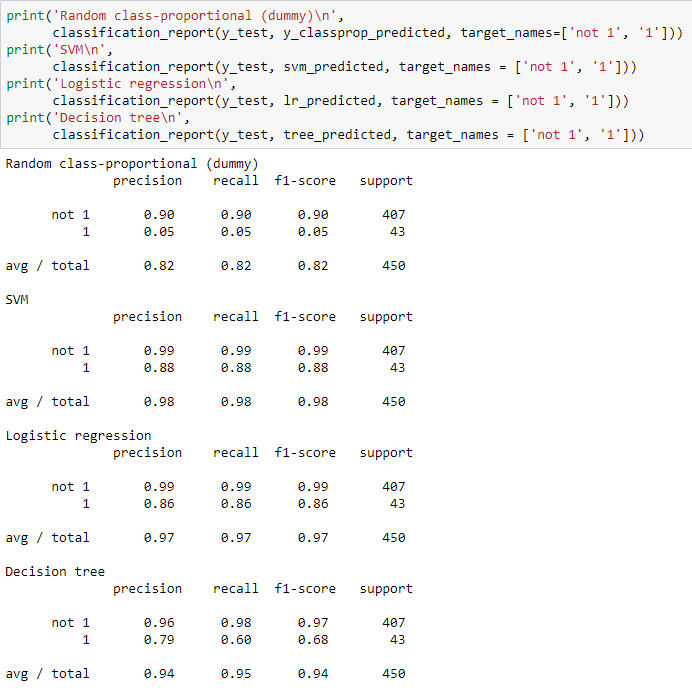
**Computing Metrics:**



Using the classification report we can also assign target names to the matrix to make it easier to understand.



The below code shows the performance and classification reports for all the methods we tried above:



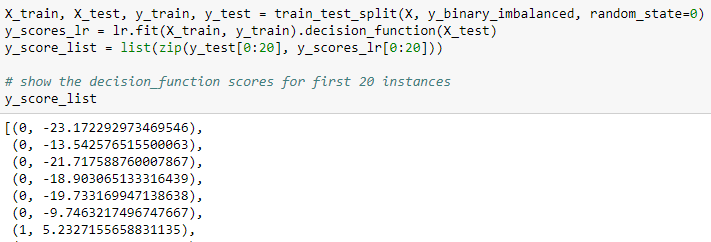
**Classifier Decision Functions:**

Most classifier is Scikit-learn can provide information about the **probability of the classification**, either with the **decision function** method or the **predict proba** method.

**Decision Function:**

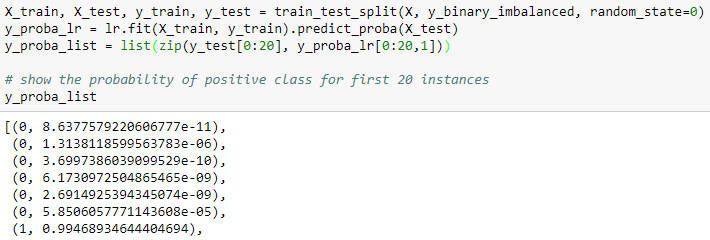
For each data entry the method returns a score of how likely the class selected was the correct one. For positive classes the score given will be positive, and for negative classes the score will be negative.

Below the “lr” is logistic regression. The code outputs the score of how confident the classifier is in its predictions.



**Predict Proba:**

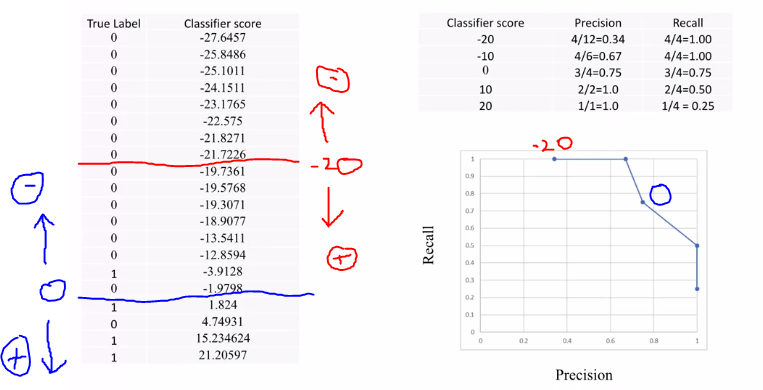
This returns a probability of how confident the model things that classification is.



Remember that these predictions of probability can still be subjects to the pains of over training. These method are used in **Precision Recall Curves**.

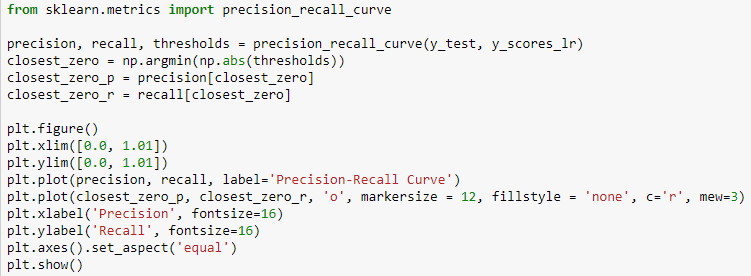
**Precision-Recall and ROC curves:**

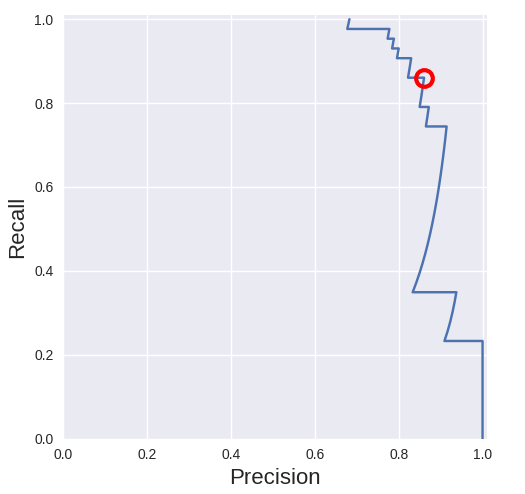
**Precision-Recall:**



The above figure shows the example of how decision functions are used to produce Precision and Recall curves. Take a classifier score of -20 and we cut the output in half with all the scores less than -20 being in the negative class and all the score above being in the + class. We can then calculate the precision and recall of these division and plot them, each time changing the classifier score boundary.

We can compute this **Precision**-**Recall** **Curve** automatically using Scikit-learn:



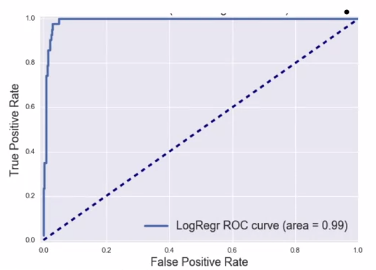


Code explained: threshold is all the decision scores of the predictions, “np.abs(thresholds)” then makes these all positive, “np.argmin(…)” then returns the index of the minimum value from the “np.abs(threshold)”. So the red circle is showing where precision and recall are equal and the highest.

**ROC (Receiver Operating Characteristic) Curves:**

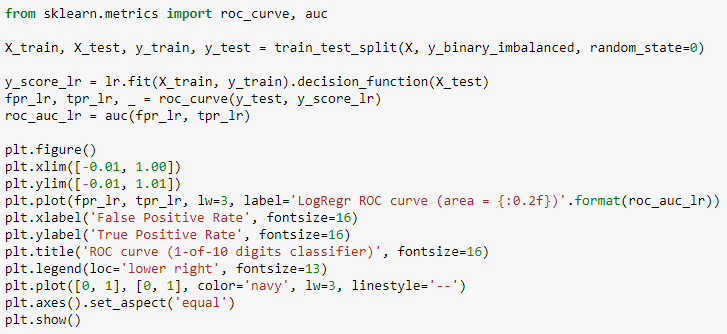
This is a very widely used visualization method that illustrates the performance of a binary classifier. It plots the **False Positive Rate** on the x-axis and the **True Positive Rate** on the y-axis, the ideal position on the axis would be in the top left-hand corner, where false positive rates would be 0 and the true positive rates would be 1.

**The curve shows how the trade-off as the decision threshold is varied for the classifier. This is the same as we in the precision recall as there we also varied the decision threshold.**



The solid line shows the models response to having its decision threshold varied. The dotted line shows the performance of a **dummy classifier** that randomly selects the output class, so we always want our model to be on the **right-hand side of this line**.

**We can determine who good a classifier is by finding the area under this curve AUC.**



The output is the figure ROC curve above.

We us the “roc\_curve” to generate the points for the plot, and we use “auc(…)” to find the area under the curve.

**More info on ROC:**

The **ROC curve** is a plot that saves us from looking at every confusion matrix generated from all the possible **threshold** changes. The y axis is recall (a high number means **less False Negatives**), and the x axis is the false positive rate (this is the fraction of all **negative instances** that the classifier **incorrectly** identifies as **positive**).

If we look at the top right of a ROC curve (1,1) this shows us a threshold that makes the model correctly classify all target values but incorrectly classify all the non-target values.

If we wanted a model that was optimised for recall we would choose a threshold that gave coordinates with the highest value of True positive rate (y-axis) and the smallest False negative rate. For the ROC above that would be (0.075,1).

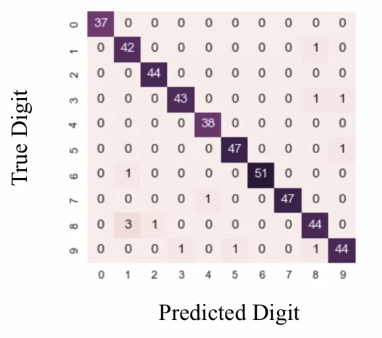
The **area under** a **ROC curve is used to compare other models to one another**, the higher the ROC the better the model is (usually).

**Multi-Class Evaluation:**

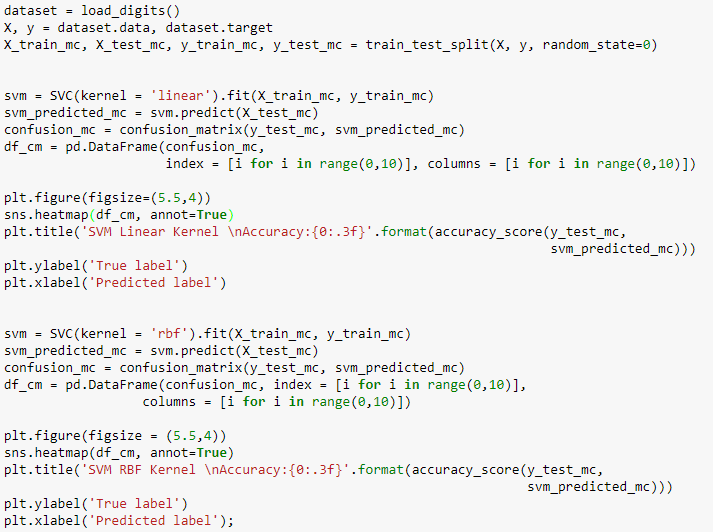
Multi-class evaluation is similar to an extension of binary classification, instead of 2 classes we have multiple classes. The output I a collection of true vs predicted binary outcomes, one for each class. Confusion matrices are especially useful in multi-classification.

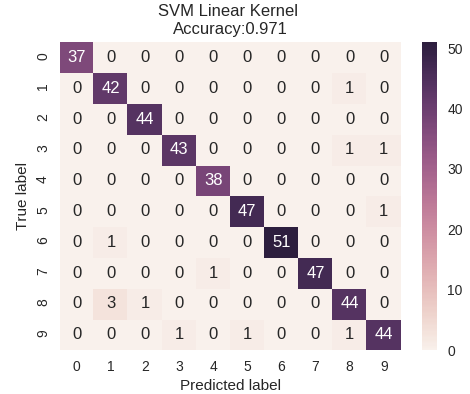
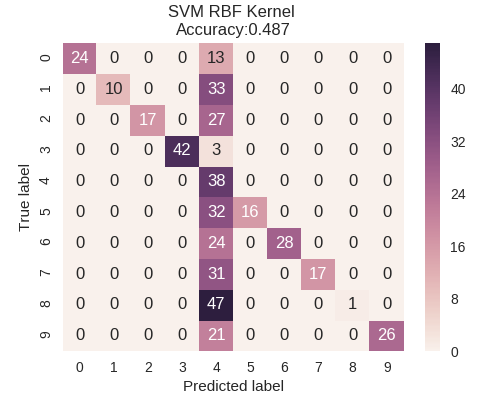
However, there are different ways to **average** multi-class results. The **support** (number of instances per class) is also important here. E.g. in the case of class imbalance.

Below is an example of a confusion matrix of the digits problem talked about above. The correct prediction of the classifier lies on the left to right diagonal.



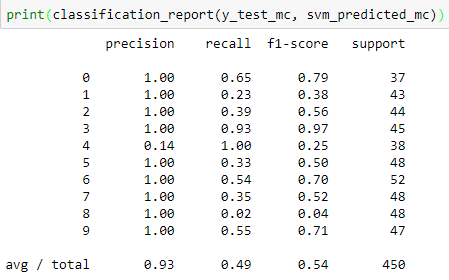
The code below looks at using a LSVM and a Radial basis function SVM to predict the digits:





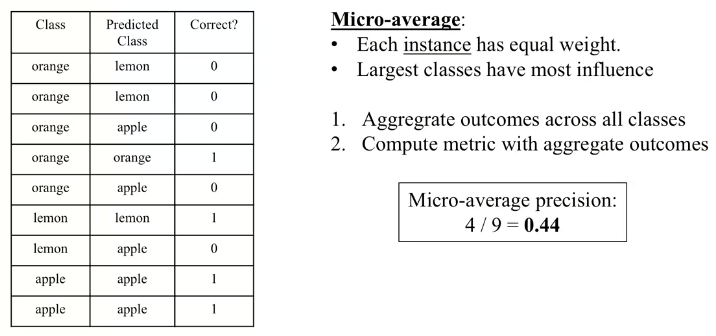
The matrix has been plotted as a heat map, this allows for quick and simple visualization of the performance. We can see that the LSVM actually performs very well, whereas the radial basis function did well apart from its over prediction of digit 4. If we only used accuracy, we wouldn’t have gotten this insight.

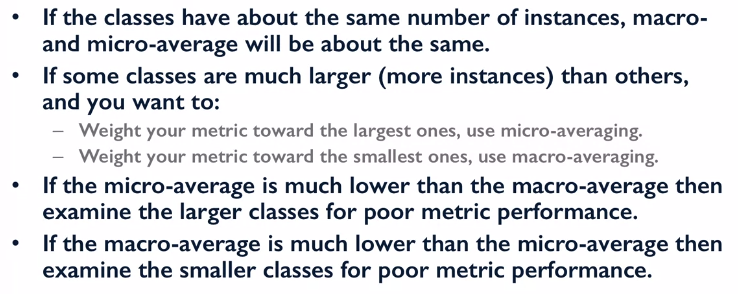
Just like in the binary classification problem, we can use a classification report to calculate metrics of the model’s performance:



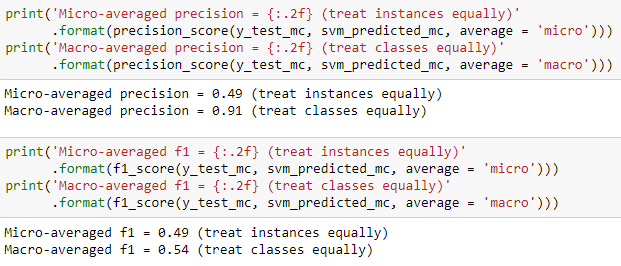
We can see that the **precision** of 4 is very low for this class as there are many **false positives**.

The **Macro average** is the overall score. E.g. if calculating the Macro precision:

In **Micro average** each instance has equal weight, so the largest class has the most influence. E.g.:



We can use code to calculate these metrics:



**Regression Evaluation:**

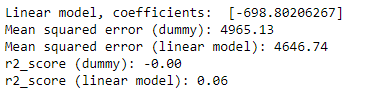
Evaluating regression models is much easier than classification, and in most cases using a simple **R^2 score** is actually enough.

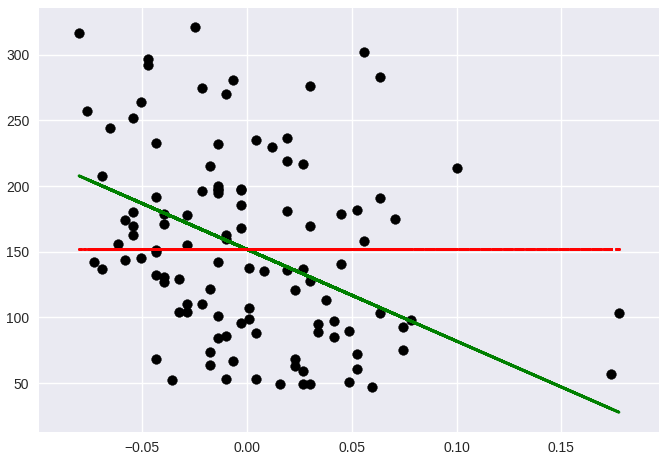
Other evaluation methods:

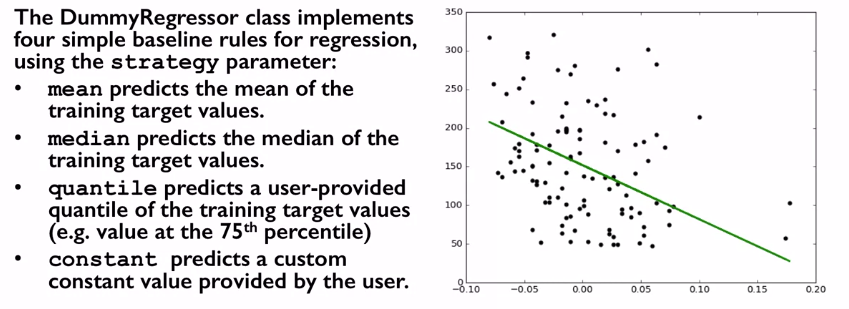
* **Mean absolute error**
* **Mean squared error**
* **Median absolute error**
* **Root meant square error**

The **median absolute error** score is good at leaving out large abnormal errors in the results.

Similarly, to classification we can use a **dummy regressor**. The following code shows the use of these evaluation methods and a dummy regressor:

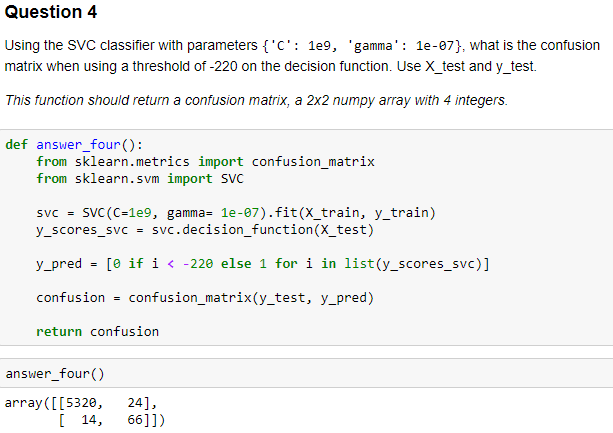






**Assignment 3 Useful Questions and Answers:**

Using the decision function and predict probability is performed after the model has been trained. We can then produce a y\_scores array that gives a score of how likely the sample is to be one class or the other. By plotting a recall-precision plot you can choose a threshold that is best suited for your model (precision recall tradeoff). If you deploy the model you need to produce a decision function/ predict probability score for the new data e.g. the y\_pred method does this below for test data, but the same principal applies to completely new data.



A more useful output would have been precision and recall. But that can be found from above.

