A^3 - Aprendizagem Automática Avançada

End to End Machine Learning Classification Project

𝔾. Marques

Working with Real Data

In Machine Learning it is essential to experiment with real data. Fortunately, there are thousands of datasets available on-line, covering all sorts of domains. Here are some places where one can get the data:

- UCI Machine Learning Repository
- Kaggle Datasets
- Amazon AWS Datasets

Or you can find listings of popular repositories in these pages:

- Wikipedia list of Machine Learning datasets
- GitHub
- Carnegie Mellon University Library

Furthermore, many Python packages such as `sklearn` or `Keras` also come with or are able to download many datasets. `TensorFlow` also provides a large collection of ready-to-use datasets. Check out TensorFlow Datasets.

Fashion MNIST

In this project we will use the fashion MNIST dataset. It consists of 70 000 gray scale, 28×28 images of ten types of clothing items. This datasets was based on the original MNIST dataset, and follows the same train/test split: 60 000 training examples and 10 000 test examples. Here are twenty examples of each of the 10 classes.

Class names:
T-shirt/Top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle boot.

Fashion MNIST

Loading the data with Keras

Keras provides a simple way to fetch and load many datasets, including the fashion-MNIST dataset:

```
fashion_minist=keras.datasets.fashion_mnist
(Xtrain,ytrain),(Xtest,ytest)=fashion_mnist.load_data()
```

`Xtrain:`	`uint8` array of shape	`(60000,28,28)`
`ytrain:`	`uint8` array of shape	`(60000,)`
`Xtest:`	`uint8` array of shape	`(10000,28,28)`
`ytest:`	`uint8` array of shape	`(10000,)`

This dataset already comes split into training and a testing set, and the examples in both sets are already shuffled for us. It is important to assert that the training examples are shuffled since may learning algorithms are sensitive to the order of the training instances and are affected is they receive in a row several examples from the same class. Furthermore, training/evaluation methodologies such as cross-validation also required the data to be shuffle to guarantee that each fold contains examples from all classes. To shuffle the data one can use the `random` module of Numpy:

```
import numpy.random as rd
idx=rd.permutation(Xtrain.shape[0])
Xtrain=Xtrain[idx]
ytrain=ytrain[idx]
```

Data Pre-processing

When dealing with many types of data such as images, it is imperative to pre-process the data before applying any Machine Learning techniques. Note that images come in `uint8` Numpy arrays and therefore we need to cast them as float. Also these images need to be converted to vectors, since this is the format used by many classification algorithms. In our case, each image will be converted into a 784 dimensional vector:

```
Xtrain=Xtrain*1.0 # convert to float
Xtrain=np.reshape(Xtrain,(60000,784))
```

A standard data normalization technique is to subtract the data mean and rescale each dimension in order to have unit standard deviation. The Sklearn `preprocessing` module come with this method and a few others:

```
m=np.sort(np.mean(Xtrain,axis=0)) # sorted data mean
print(m[:5],m[-5:])  # lowest and highest means
  [0.  0.01 0.01 0.02 0.02] [159.2 159.5 160.7 161.4 161.9]
v=np.sort(np.std(Xtrain,axis=0)) # sorted standard deviation
print(v[:5],v[-5:])  # lowest and highest values
  [0.09 0.25 0.57 0.77 0.93] [101.6 102.3 102.8 103.6 103.7]
import sklearn.preprocessing as pp
sc=pp.StandardScaler().fit(Xtrain)
XtrainS=sc.transform(Xtrain)
```

Now each data dimension has zero mean and unit variance. This type of normalization works well with most data but not if it contains noisy values and outliers. In this situation one should choose the `RobustScaler` which handles better this type of noisy data.

Binary Classification

To start with, let's deal with a simple problem of identifying a single class, for example, the class, "Dress" (class number 3). First we need to create the target vectors for this problem:

```
ytrain_b=(ytrain==3)*1
ytest_b=(ytest==3)*1
```

Now we can choose a classifier. Let us pick a **Stochastic Gradient Descent** classifier (`SGDClassifier`) which has the advantage of dealing well with large datasets. Let us also use the whole training set to train it:

```
from sklearn.linear_model import SGDClassifier
sgd=SGDClassifier().fit(XtrainS,ytrain_b)
print(100*sgd.score(XtestS,ytest_b)) # pre-processed test data + labels
94.90
```

The accuracy may seem high but one should also note that classifying all the test images as negative would get an accuracy of 90%. This is why accuracy is generally not the chosen performance measure, specially when dealing with skewed datasets. It is more informative to look at the confusion matrix and at the precision, recall, and f-score values.

	$\hat{P}$	$\hat{N}$
P	862	138
N	372	8628

$$\text{precision} = \frac{862}{862+372} = 0.699 \qquad \text{recall} = \frac{862}{862+138} = 0.862$$

$$\text{accuracy} = 94.9\% \qquad \text{f-score} = 2\frac{\text{prec.} \times \text{rec.}}{\text{prec.} + \text{rec.}} = 0.772$$

Model Fine-Tuning

Another factor that directly influences model performances is the choice of the model's **hyper-parameters**. Hyper-parameters are not a part of the model but control how the models parameters are calculated during training. These hyper-parameters have to be set before hand, and the wrong choice of values can significantly deteriorate the model's performance. Nevertheless, setting the hyper-parameters values manually is, in many cases, an impossible task due to the huge number combinations.

One solution is to test all the combinations of the hyper-parameters values using a strategy called **grid search**. One drawback of this approach is its computational cost, so one has choose wisely which hyper-parameters and values to test.

A final important point about adjusting hyper-parameters: in order to find which values work best, one has to test the model with data that was not used for training. The test set is off-limits: if we use this set, then it is part (indirectly) of the training process. We have to split the training data in two. Part will be the training set and the other the **validation set**.

Model Fine-Tuning

Like many other Sklearn models, the `SGDClassifier` has several hyper-parameters that can be user-defined (but most will work well with their default values). Let's adjust one of our model hyper-parameters, `alpha` (regularization parameter). First we need build a training and validation set and define the hyper-parameter values before doing the grid search.

```
from sklearn.model_selection import train_test_split as ttsplit
Xtrain0,Xvalid,ytrain0,yvalid=ttsplit(XtrainS,ytrain_b,test_size=0.4)
alphaArray=np.arange(1e-5,1e-3,5e-5) # 20 values
```

Now we can proceed with the grid search.

```
topScore=0.
for a in alphaArray:
  sgd=SGDClassifier(penalty='l2',alpha=a).fit(Xtrain0,ytrain0)
  currentScore=sgd.score(Xvalid,yvalid)
  if currentSocre>topScore:
    currentSocre=topScore
    bestAlpha=a
```

(best `alpha`: 0.00051)

Re-train with the whole training set

```
sgd=SGDClassifier(penalty='l2',alpha=bestAlpha).fit(Xtrain,ytrain_b)
```

		$\hat{P}$	$\hat{N}$		
Test set measures:	P	814	186	precision=0.7615	recall=0.814
	N	255	8745	accuracy= 95.59%	f-score=0.787

Testing Methodologies

To evaluate the performance of a classifier one needs to test it with data it has never seen (the testing set). This tells us how the classifier will behave with new data. Also testing with new data also enables to see if the model is **overfitting** the training data.
Having the evaluation depending on a single test set has its limitations: we only get a single estimate of the performance measures and we do not know its variability. A simple solution is to repeat the training and testing with different datasets, which yields several performance estimates instead of a single one. An elegant way of doing this is through K-Fold cross-validation.

K-Fold Cross-Validation

This is a technique to evaluate the models with limited training data. In K-fold cross-validation, the labeled data is divided into K groups or folds, and all but one are used for training and the remaining for testing. This procedure is repeated over all the K folds.

- All the data is tested.
- We obtain K performance measures which permits to assess the variance of the estimation.
- The drawback is the increase of the computational load. For complex models, like DNN, cross-validation may not be an option due to computational limitations.

K-fold cross-validation only gives us a more precise estimate of the performance of the model compared to a simple train/test split strategy. Cross validation can be used for model fine-tuning but it does not give us a single model. For model deployment, one has to re-train with the whole dataset.

Testing Methodologies

K-Fold Cross-Validation

Sklearn has several functions under the module `model_selection` to deal with cross-validation. In these functions, the division process of the data into folds is done internally and the user has only to specify a few parameters. Such functions include `cross_val_score` and `cross_val_predict` for model evaluation, and `GridSearchCV` for model fine-tuning.

The cross-validation strategies can be specified with iterators, such as the following:

- `KFold` and `StratifiedKFold`:
  `KFold` is the standard cross-validation strategy To shuffle the data you can set the `shuffle` parameter to `True` and specify the number of folds with `n_splits`. In the stratified version of the iterator, the overall class distribution is preserved (approximately) in each fold. It is wise to use this strategy, particularly when dealing with skewed datasets.

  Next a binary classifier is trained and tested with stratified 10-fold cross-validation:

```
import sklearn.model_selection as ms
cv=ms.StratifiedKFold(n_splits=10,shuffle=True,random_state=42)
sgd=SGDClassifier(penalty='l2',alpha=0.00051)
sc=ms.cross_val_score(sgd,XtrainS,ytrain_b,cv=cv)
print(np.round(1e4*sc)/100)
96.25 96.25 96.47 96.82 96.3 96.37 96.13 96.3 96.15 96.22
```

  (mean: 96.33)

Testing Methodologies

K-Fold Cross-Validation

Sklearn has several functions under the module `model_selection` to deal with cross-validation. In these functions, the division process of the data into folds is done internally and the user has only to specify a few parameters. Such functions include `cross_val_score` and `cross_val_predict` for model evaluation, and `GridSearchCV` for model fine-tuning.

The cross-validation strategies can be specified with iterators, such as the following:

- `ShuffleSplit` and `StratifiedShuffleSplit`:
  This evaluation approach randomly divides the data into a training and test sets, and repeats the process K times. This is similar to K-fold cross-validation, but there is no certainty that the whole dataset is covered during testing.

  Next a binary classifier is trained and tested with stratified shuffle-split (with 10 splits):

```
import sklearn.model_selection as ms
cv=ms.StratifiedShuffleSplit(n_splits=10,test_size=1/10)
sgd=SGDClassifier(penalty='l2',alpha=0.00051)
sc=ms.cross_val_score(sgd,XtrainS,ytrain_b,cv=cv)
print(np.round(1e4*sc)/100)
96.35 96.4 96.45 96.2 96.38 96.55 96.42 96.33 96.22 96.57
```

  (mean: 96.39)

Testing Methodologies

Sklearn also has a function that performs a grid search with cross-validation to fine-tune the model's hyper-parameters. For example, lets test a new binary classification model with two hyper-parameters to adjust. The model is a **Random Forest Classifier**. The two hyper-parameters are: `n_estimators` that controls the number of trees in the forest, and `max_depth` the maximum depth of the leaves. We have to instantiate the classifier and a dictionary with the hyper-parameter names and values we want to test.

```
from sklearn.model_selection import GridSearchCV,StratifiedKFold
from sklearn.ensemble import RandomForestClassifier
rfClass=RandomForestClassifier()
Ne=np.arange(75,151,25)
depth=np.arange(2,6)
parD={'n_estimators':Ne,'max_depth':depth}
cv=StratifiedKFold(n_splits=3,shuffle=True,random_state=42)
grs=GridSearchCV(rfClass,param_grid=parD,cv=cv).fit(XtrainS,ytrain_b)
```

Re-train with the whole training set

```
p=grs.best_params_ #{'max_depth':  30, 'n_estimators':  125}
sgd=RandomForestClassifier(n_estimators=p['n_estimators'],
    max_depth=p['max_depth']).fit(Xtrain,ytrain_b)
```

Test set measures:		$\hat{P}$	$\hat{N}$		
	P	715	285	precision=0.896	recall=0.715
	N	83	8917	accuracy= 96.32%	f-score=0.795

Testing Methodologies

Sklearn also has a function that performs a grid search with cross-validation to fine-tune the model's hyper-parameters. For example, lets test a new binary classification model with two hyper-parameters to adjust. The model is a **Random Forest Classifier**. The two hyper-parameters are: `n_estimators` that controls the number of trees in the forest, and `max_depth` the maximum depth of the leaves. We have to instantiate the classifier and a dictionary with the hyper-parameter names and values we want to test.

```
from sklearn.model_selection import GridSearchCV,StratifiedKFold
from sklearn.ensemble import RandomForestClassifier
rfClass=RandomForestClassifier()
Ne=np.arange(75,151,25)
depth=np.arange(2,6)
parD={'n_estimators':Ne,'max_depth':depth}
cv=StratifiedKFold(n_splits=3,shuffle=True,random_state=42)
grs=GridSearchCV(rfClass,param_grid=parD,cv=cv).fit(XtrainS,ytrain_b)
```

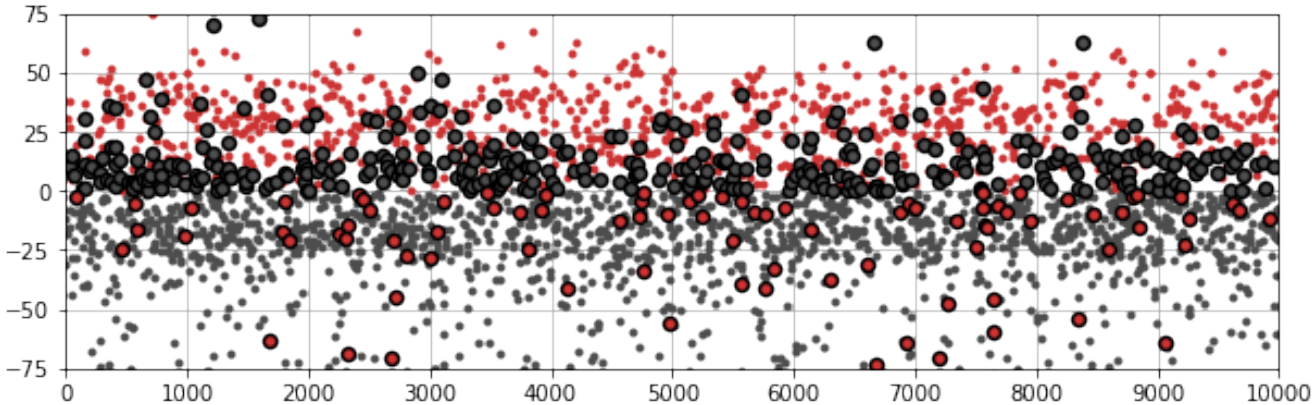
Be careful not to pick too many combinations to test. In this case, $3 \times 4 \times 4 = 48$ models are trained and tested.

Hint: you can use grid search with `cross_val_score` to evaluate the performance on more than one test set (at an additional computational expense).

Model Calibration

In binary classification problems, it is important to calibrate the model to find the optimal operational point for the task at hand. For this, **we do not re-train the model**. This is done by adjusting the decision threshold of the model so that it outputs more "conservative" or more "liberal" predictions (more 0s or more 1s). In order to do that, one has to access the output of the classifier before it as has been converted to 0s or 1s. This can be done through the method `decision_function` present in most Sklearn models. The next instructions calculate the model scores and predictions and obtain the confusion matrix for the test set:

```
from sklearn.linear_model import SGDClassifier
from sklearn.metrics import confusion_matrix
sgd=SGDClassifier().fit(XtrainS,ytrain_b)
ytest_score=sgd.decision_function(XtestS)
ytest_p=sgd.predict(XtestS)
CofMat=confusion_matrix(ytest_b,ytest_p)
```

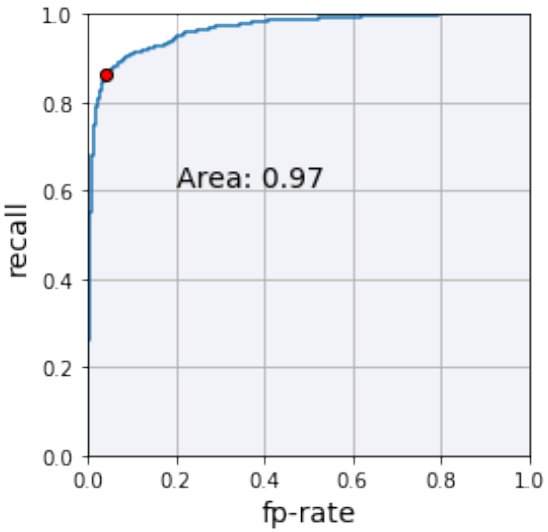


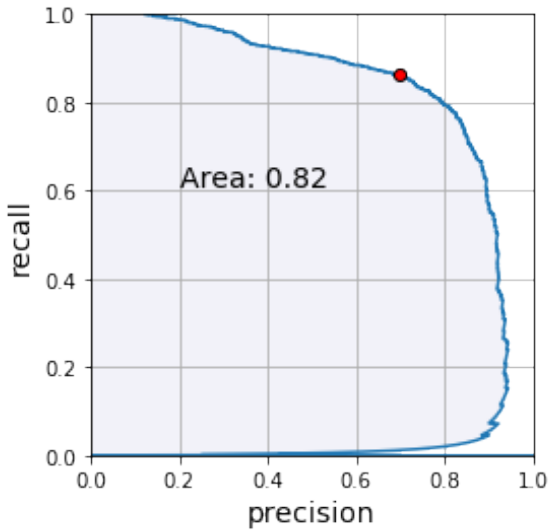
Plot of the scores of the test set instances.
Gray points: negatives, red points: positives, bold points: errors.

Model Calibration

Let us visualize the precision/recall and ROC curves and plot in these the current operational point of our model. These can be obtained with the sub-module metrics of Sklearn:

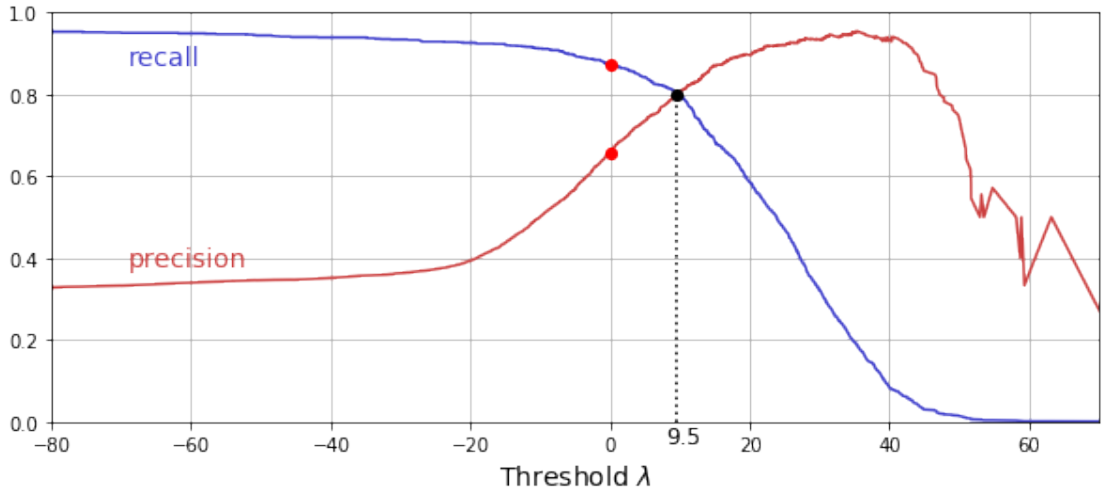
```
import sklearn.metrics as skm
fpr,rec,thresh=skm.roc_curve(ytest_b,ytest_score)
area1=skm.roc_auc_score(ytest_b,ytest_score)
prec,rec,thresh=skm.precision_recall_curve(ytest_b,ytest_score)
area2=skm.average_precision_score(ytest_b,ytest_score)
```





Model Calibration

The calibration of binary models depends on the task at hand and the costs of making a false prediction may vary. In some situations one may want to minimize the number of false positives while in others reduce the number of false negatives. For this problem, let us find the operational points that yields the highest values for both the precision and recall.This is equivalent to saying that the cost of the two types of errors is the same. For that we have to see how the curves vary in terms of the thresholds and find where they intercept.

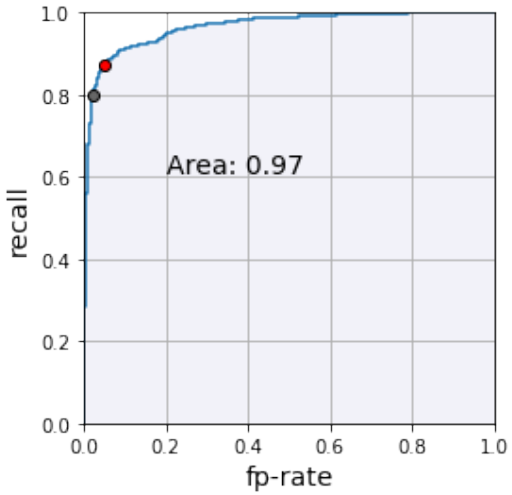


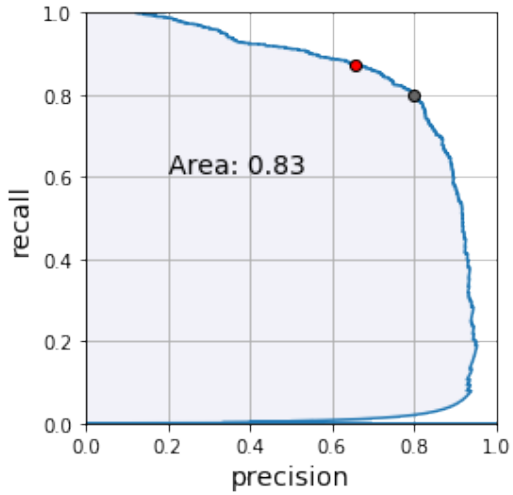
New confusion matrix (with $\lambda = 9.5$):

	$\hat{P}$	$\hat{N}$
P	799	201
N	201	8799

Model Calibration

The calibration of binary models depends on the task at hand and the costs of making a false prediction may vary. In some situations one may want to minimize the number of false positives while in others reduce the number of false negatives. For this problem, let us find the operational points that yields the highest values for both the precision and recall.This is equivalent to saying that the cost of the two types of errors is the same. For that we have to see how the curves vary in terms of the thresholds and find where they intercept.





New confusion matrix (with $\lambda = 9.5$):

	$\hat{P}$	$\hat{N}$
P	799	201
N	201	8799

Model Comparison

So far we have trained two classifiers on a binary problem: the `SGDClassifier` and the `RandomForestClassifier`. ROC and precision/recall curves are a simple way to visually compare the performances of the two classifiers.

Let us re-train the two classifiers with the previously found parameters that yielded the best results:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import SGDClassifier
rfC=RandomForestClassifier(max_depth=5,n_estimators=75)
rfC.fit(XtrainS,ytrain_b)
ytest_p1=rfC.predict(XtestS)
sgd=SGDClassifier(penalty='l2',alpha=0.00051).fit(XtrainS,ytrain_b)
ytest_p2=sgd.predict(XtestS)
```

Here are the results on the test set:

		$\hat{P}$	$\hat{N}$		
Random Forest:	P	634	366	precision=0.893	recall=0.634
	N	76	8924	accuracy= 95.6%	f-score=0.742

		$\hat{P}$	$\hat{N}$		
`SGDClassifier`:	P	820	180	precision=0.769	recall=0.82
	N	246	8754	accuracy= 95.7%	f-score=0.794

Model Comparison

So far we have trained two classifiers on a binary problem: the `SGDClassifier` and the `RandomForestClassifier`. ROC and precision/recall curves are a simple way to visually compare the performances of the two classifiers.

Let us re-train the two classifiers with the previously found parameters that yielded the best results:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import SGDClassifier
rfC=RandomForestClassifier(max_depth=5,n_estimators=75)
rfC.fit(XtrainS,ytrain_b)
ytest_p1=rfC.predict(XtestS)
sgd=SGDClassifier(penalty='l2',alpha=0.00051).fit(XtrainS,ytrain_b)
ytest_p2=sgd.predict(XtestS)
```

Looking at the four performances measures of both classifiers, the stochastic gradient descent classifier outperforms the random forest classifier in 3 out of the four measures. Nevertheless, looking at just these numbers can be misleading. A better sense of each classifiers performance is obtained looking at the precision/recall and ROC curves.

Model Comparison

So far we have trained two classifiers on a binary problem: the `SGDClassifier` and the `RandomForestClassifier`. ROC and precision/recall curves are a simple way to visually compare the performances of the two classifiers.

Let us take a look at the precision/recall and ROC curves for both classifiers. As previously mentioned, most Sklearn classifiers have a method called `decision_function` that can be used to obtain the curves, but the `RandomForestClassifier` is an exception. Nevertheless, we can use another method present in most classifiers (random forest included) called `predict_proba`. This method returns a matrix $N \times c$, where N is the number of instances and c the number of classes. In this problem $N = 10000$ and $c = 2$ (for the test set). For the random forest classifier, we need to pick the second column of this output matrix (the positive class).

Next are the commands to obtain the curves:

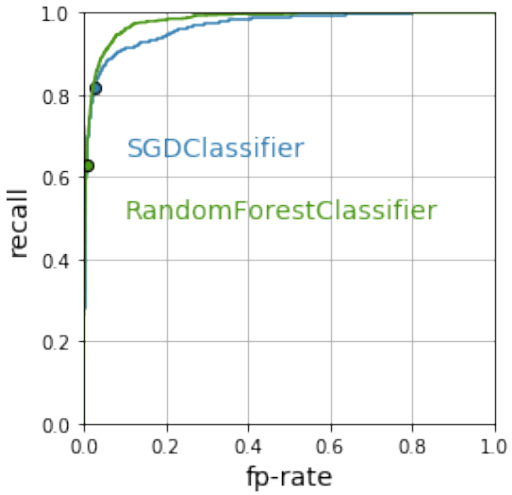
```
import sklearn.metrics as skm
# RandomForestClassifier
ytest_score1=rfC.predict_proba(XtestS)[:,1] # get the 2nd column
prec,rec,thresh=skm.precision_recall_curve(ytest_b,ytest_score1)
fpr,rec,thresh=skm.roc_curve(ytest_b,ytest_score1)

# SGDClassifier
ytest_score2=sgd.decision_function(XtestS)
prec,rec,thresh=skm.precision_recall_curve(ytest_b,ytest_score2)
fpr,rec,thresh=skm.roc_curve(ytest_b,ytest_score2)
```

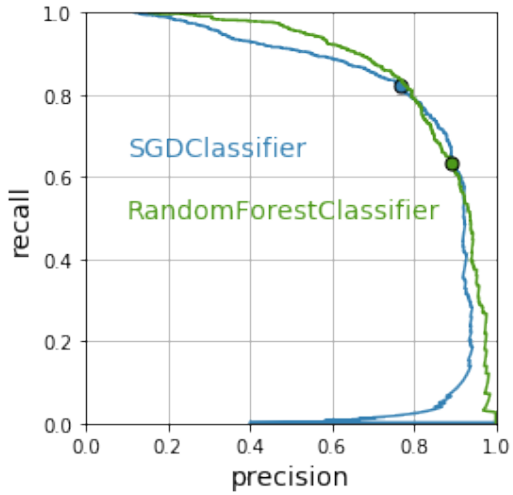
Model Comparison

So far we have trained two classifiers on a binary problem: the `SGDClassifier` and the `RandomForestClassifier`. ROC and precision/recall curves are a simple way to visually compare the performances of the two classifiers.

Let us take a look at the precision/recall and ROC curves for both classifiers. As previously mentioned, most Sklearn classifiers have a method called `decision_function` that can be used to obtain the curves, but the `RandomForestClassifier` is an exception. Nevertheless, we can use another method present in most classifiers (random forest included) called `predict_proba`. This method returns a matrix $N \times c$, where N is the number of instances and c the number of classes. In this problem $N = 10000$ and $c = 2$ (for the test set). For the random forest classifier, we need to pick the second column of this output matrix (the positive class).

Here are the curves for the two classifiers:





Looking at the plots, it is now clear that the random forest classifier has a better performance than the stochastic gradient one, but needs to be calibrated to have a better operational point.

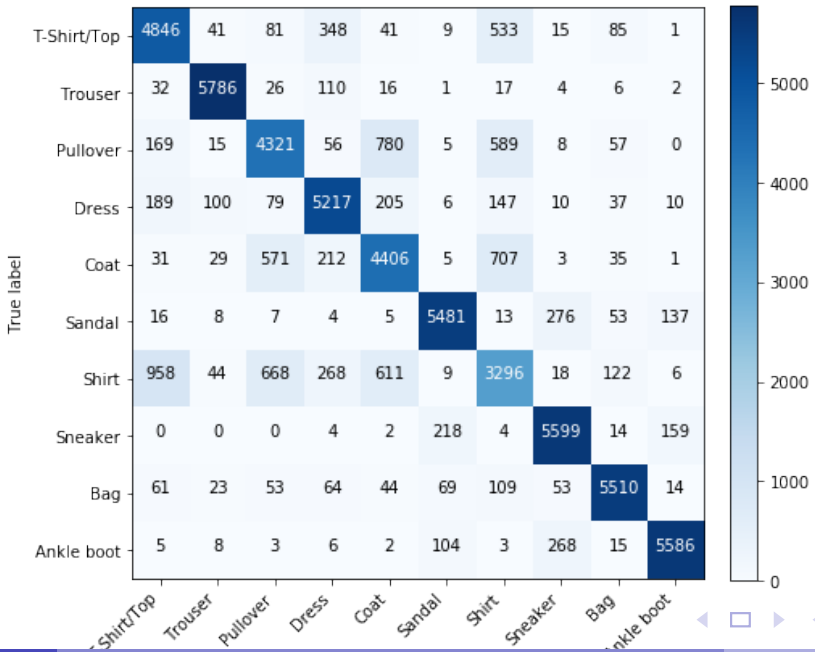
Multi-Class Classification

In multi-class classification each example belongs to one of a number of pre-defined, mutually exclusive, classes. Let us see how a linear **Support Vector Machine** handles the fashion-MNIST dataset.

```
from sklearn.svm import LinearSVC
model=LinearSVC()
cv=StratifiedKFold(n_splits=4,shuffle=True,random_state=42)
score=cross_val_score(model,XtrainS,ytrain,cv=cv)
y1p=cross_val_predict(model,XtrainS,ytrain,cv=cv)
```

Next let's see how the classifier did on the validation folds (*i.e.* the whole training set).
Accuracy per fold: 83.85%, 83.64%, 83.67%, 83.41%
Mean Accuracy: 83.64%

Confusion matrix:



Multi-Class Classification

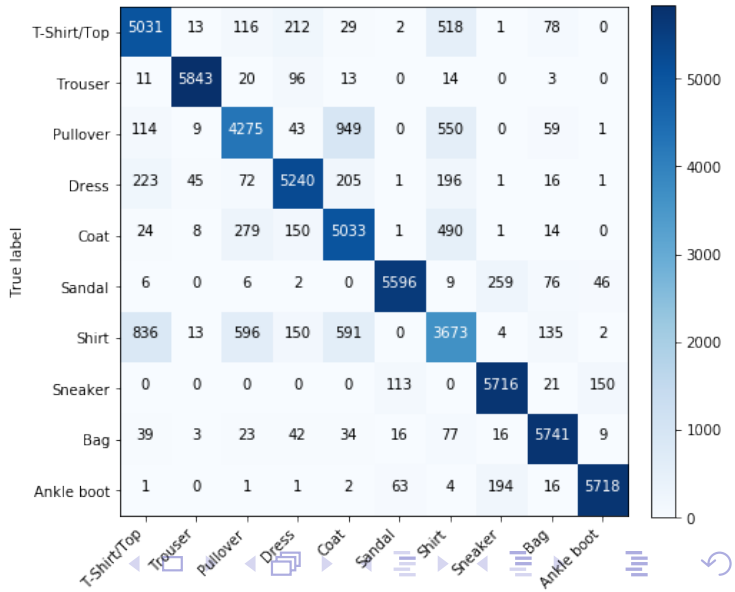
In multi-class classification each example belongs to one of a number of pre-defined, mutually exclusive, classes. Let us see how a linear **Support Vector Machine** handles the fashion-MNIST dataset.

```
from sklearn.svm import LinearSVC
model=LinearSVC()
cv=StratifiedKFold(n_splits=4,shuffle=True,random_state=42)
score=cross_val_score(model,XtrainS,ytrain,cv=cv)
y1p=cross_val_predict(model,XtrainS,ytrain,cv=cv)
```

Let's check the performance on the test data. For that, we need to re-train a new classifier with the whole training set. It is also important to analyze the performance on the training set. Large discrepancies in performances between the training and test sets may indicate that the model is over-fitting the training data.

```
model=LinearSVC().fit(XtrainS,ytrain)
y1p=model.predict(XtrainS)
y2p=model.predict(XtestS)
```

Training set accuracy: 86.44%



Multi-Class Classification

In multi-class classification each example belongs to one of a number of pre-defined, mutually exclusive, classes. Let us see how a linear **Support Vector Machine** handles the fashion-MNIST dataset.

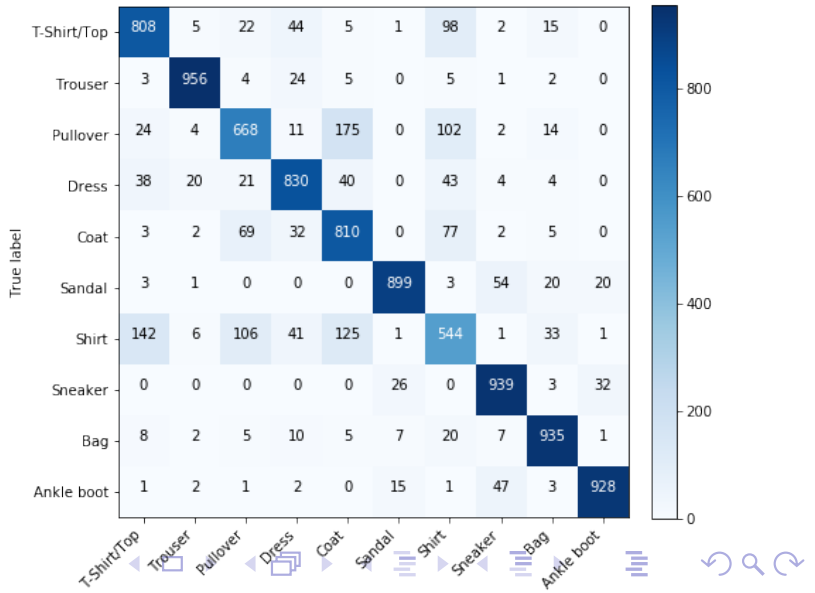
```
from sklearn.svm import LinearSVC
model=LinearSVC()
cv=StratifiedKFold(n_splits=4,shuffle=True,random_state=42)
score=cross_val_score(model,XtrainS,ytrain,cv=cv)
y1p=cross_val_predict(model,XtrainS,ytrain,cv=cv)
```

Let's check the performance on the test data. For that, we need to re-train a new classifier with the whole training set. It is also important to analyze the performance on the training set. Large discrepancies in performances between the training and test sets may indicate that the model is over-fitting the training data.

```
model=LinearSVC().fit(XtrainS,ytrain)
y1p=model.predict(XtrainS)
y2p=model.predict(XtestS)
```

Test set accuracy: 83.17%

Note that this accuracy is a little less that the predicted by the validation set (but still pretty close). On the other hand the training set accuracy is more than 3 points higher than the test set accuracy.



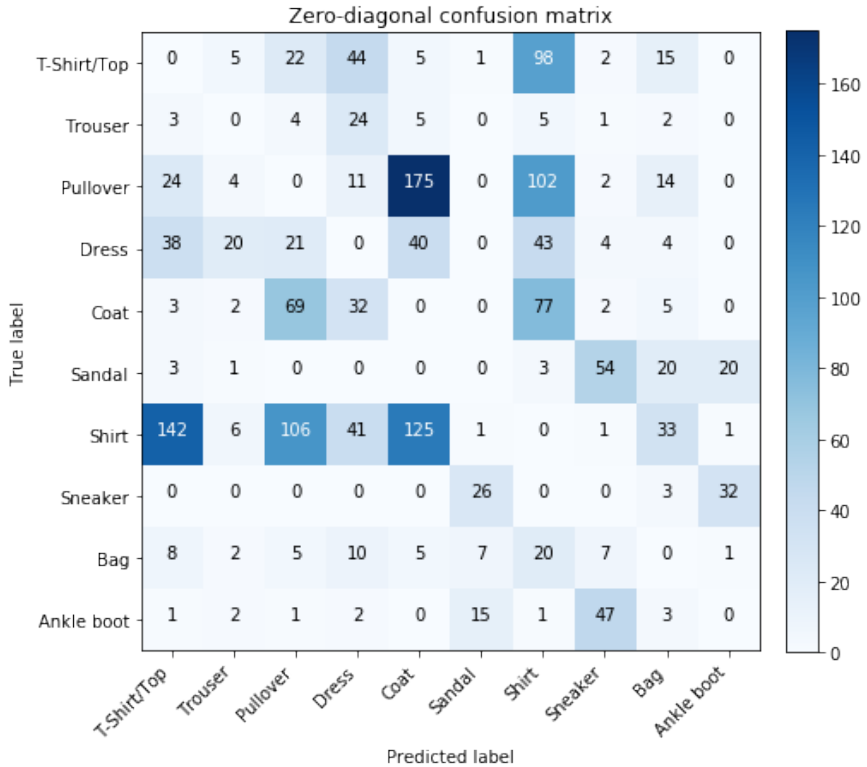
Multi-Class Classification

Error Analysis:

Analyzing the classification errors often gives an insight on ways to improve its performance. One can check the errors per class to see if one or more classes is performing poorly. Another way is to check if there are salient error patterns present in the confusion matrix. For this purpose, it is better to fill the diagonal of the matrix (the correct predictions) with zeros, so that the errors stand out.

Per class error percentages:

T-Shirt/Top	19.2%
Trouser	4.4%
Pullover	33.2%
Dress	17.0%
Coat	19.0%
Sandal	10.1%
Shirt	45.6%
Sneaker	6.1%
Bag	6.5%
Ankle boot	7.2%



Multi-Class Classification

Error Analysis:

Analyzing the classification errors often gives an insight on ways to improve its performance. One can check the errors per class to see if one or more classes is performing poorly. Another way is to check if there are salient error patterns present in the confusion matrix. For this purpose, it is better to fill the diagonal of the matrix (the correct predictions) with zeros, so that the errors stand out.

Looking at the individual class errors and the zero-diagonal confusion matrix, several patterns emerge:

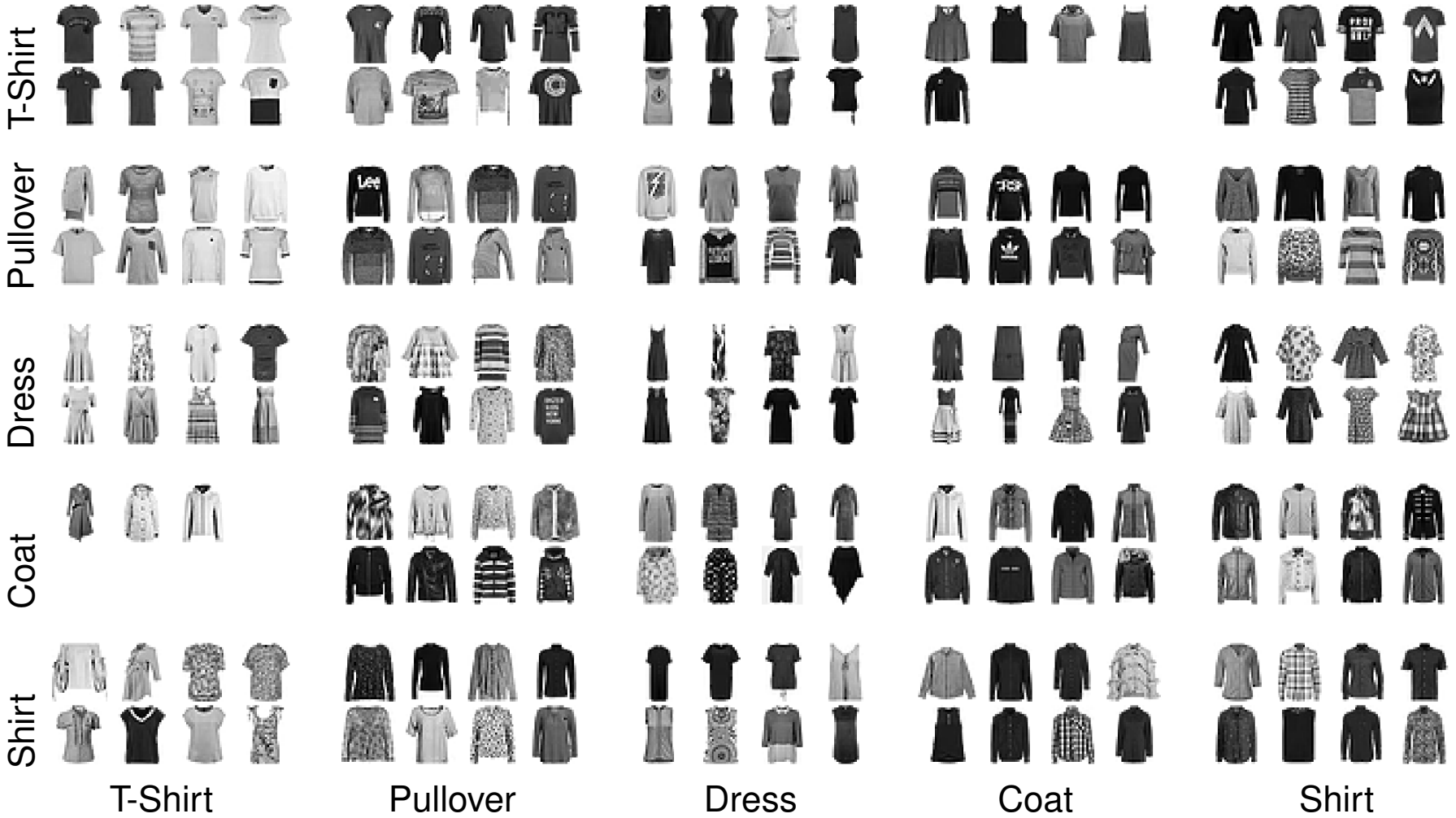
- The highest number of errors belongs to the class “Pullover” being misclassified as “Coat” (175 examples).
- The “Shirt” class has the highest error percentage (45.6%).
- There a significant amount of confusion between the dress-wear classes: “T-Shirt”, “Pullover”, “Dress”, “Coat” and “Shirt”.
- There a significant amount of confusion between the shoe-wear classes: “Sandal”, “Sneaker” and “Ankle boot”.

The errors seem to make some sense since the errors are within the dress-wear classes, and the shoe-wear classes. Looking at the individual errors can give us further insight in what the classifier is doing and why it is failing, but this process is time consuming and unfeasible (at least systematically) in most situations. In this concrete problem, we would have to analyze 1683 error images!

Multi-Class Classification

Error Analysis:

Let us look at some of the errors from the dress-wear classes:



Multi-Class Classification

Error Analysis:

Let us look at some of the errors from the shoe-wear classes: