Video #1

Introduction

Recap from MLT

- Logistic regression is the **workhorse** of machine learning.
- Before deep learning era, logistic regression was **the default choice** for solving real life classification problems with hundreds of thousands of features.
- It works in binary, multi-class and multi-label classification set ups.

Imports

In this notebook we solve the same problem of recognizing Handwritten digits using Logistic regression model.

```
# Common imports
import numpy as np
from pprint import pprint
from tempfile import mkdtemp
from shutil import rmtree

# to make this notebook's output stable across runs
np.random.seed(42)

#sklearn specific imports
# Dataset fetching


# Feature scaling
from sklearn.preprocessing import MinMaxScaler

# Pipeline utility
from sklearn.pipeline import make_pipeline

# Classifiers: dummy, logistic regression (SGD and LogisticRegression)
# and least square classification
from sklearn.dummy import DummyClassifier
from sklearn.linear_model import SGDClassifier,RidgeClassifier, LogisticRegression,Logisti

# Model selection
from sklearn.model_selection import cross_validate,RandomizedSearchCV,GridSearchCV,cross_v
from sklearn.model_selection import learning_curve

# Evaluation metrics
```

```
from sklearn.metrics import log_loss
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import precision_score, recall_score, classification_report
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import auc,roc_curve,roc_auc_score

# scipy
from scipy.stats import loguniform

# To plot pretty figures
%matplotlib inline
from  matplotlib import pyplot as plt
import seaborn as sns

# global settings
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)
mpl.rc('figure',figsize=(8,6))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-e69497263d72> in <module>()
     43
     44 # global settings
---> 45 mpl.rc('axes', labelsize=14)
     46 mpl.rc('xtick', labelsize=12)
     47 mpl.rc('ytick', labelsize=12)

NameError: name 'mpl' is not defined
```

SEARCH STACK OVERFLOW

```
# Ignore all warnings (convergence..) by sklearn
def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn
```

Note on the classification steps

Each classifier implemented for addressing this problem has the following steps:

- Preprocessing
- Classification
- Train with cross validation
- [Optional] Hyper-parameter tuning
- Performance evaluation

Handwritten Digit Classification

- We are going to repeat the digit recognition task with the following classifiers
    1. SGD classifier.
    2. Logistic Regression.
    3. Ridge Classifier.
- We make use of same **MNIST** dataset.

```
from sklearn.datasets import fetch_openml

X_pd,y_pd= fetch_openml('mnist_784',version=1,return_X_y=True)

# convert to numpy array
X = X_pd.to_numpy()
y = y_pd.to_numpy()
```

Split the dataset into training and testing set.

```
x_train,x_test,y_train,y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

Pre-Processing

- **Unlike** perceptron, where scaling the range is optional (but recommended), sigmoid requires scaling the feature range between 0 to 1.
- Contemplate the consequences if we don't apply the scaling operation on the input datapoints.
- Note: **Do not** apply mean centering as it removes zeros from the data, however, zeros should be zeros in the dataset
- Since we already visualized the samples in the dataset and know sufficient details, we are going to use `pipeline` to make the code compact.

Binary Classification : 0-Detector

- Let us start with a simple classification problem, that is, **binary classification**.
- Since the original label vector contains 10 classes, we need to modfiy the number of classes to 2. Therefore, **the label '0' will be changed to '1' and all other labels (1-9) will be changed to '0'**.
  (Important: for perceptron we set the negative labels to -1)

```
# intialize new variable names with all -1
```

```
y_train_0 = np.zeros((len(y_train)))
y_test_0 = np.zeros((len(y_test)))

# find indices of digit 0 image
indx_0 = np.where(y_train =='0') # remember original labels are of type str not int
# use those indices to modify y_train_0&y_test_0
y_train_0[indx_0] = 1
indx_0 = np.where(y_test == '0')
y_test_0[indx_0] = 1
```

Sanity check❁

- Let's display the elements of `y_train` and `y_train_0` to verify whether the labels are properly modified.

```
num_images = 9 # Choose a square number
factor = np.int(np.sqrt(num_images))
fig,ax = plt.subplots(nrows=factor,ncols=factor,figsize=(8,6))
idx_offset = 0 # take "num_images" starting from the index "idx_offset"
for i in range(factor):
  index = idx_offset+i*(factor)
  for j in range(factor):
    ax[i,j].imshow(X[index+j].reshape(28,28),cmap='gray')
    ax[i,j].set_title('Label:{0}'.format(str(y_train_0[index+j])))
    ax[i,j].set_axis_off()
```

Basline Models

- We already know that the basline model would produce an accuracy of 90.12%.

[ ] ↳ *1 cell hidden*

SGD Classifier

Before using LogisticRegression for Binary classification problem, it will be helpful to recall important concepts and equations covered in the technique course.

Recap

Let us quickly recap various components in the general settings:

1. **Training data**: (features, label) or $(\mathbf{X}, y)$, where label y is a **discrete** quantity from a finite set. **Features** in this case are pixel values of an image.
2. **Model :**

$$\begin{aligned} z &= w_0 x_0 + w_1 x_1 + w_2 x_2 + \ldots + w_m x_m \\ &= \mathbf{w}^T \mathbf{x} \end{aligned}$$

and passing it through the sigmoid non-linear function (or Logistic function)

$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+exp(-z)}$$

3. **Loss function:** We use Binary cross entropy loss.

$$J(\mathbf{w}) = -\frac{1}{n}\sum \left[y^{(i)} \log(h_{\mathbf{w}}(\mathbf{x}^{(i)})) + (1-y^{(i)})(1-\log(h_{\mathbf{w}}(\mathbf{x}^{(i)}))) \right]$$

4. **Optimization:**
   Gradient Descent

- Let's look into the paramters of the `SGDClassifier()` estimator implemented in sklearn:
  `class sklearn.linear_model.SGDClassifier(loss='hinge', *, penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=1000, tol=0.001, shuffle=True, verbose=0, epsilon=0.1, n_jobs=None, random_state=None, learning_rate='optimal', eta0=0.0, power_t=0.5, early_stopping=False, validation_fraction=0.1, n_iter_no_change=5, class_weight=None, warm_start=False, average=False)`.
  - Setting the loss parameter to `'loss=log'` makes it a **logistic regression classifier**. We may refer to documentation for more details on the `SGDClassifier` class.

Training without regularization

- Let's instatiate an object of SGDClassifier.
- Since we want to plot learning curve of training, we set `max_iter = 1` and `warm_start=True`.
- In addition, we are using constant learning rate throughout the training. For that we set `learning_rate='constant'` and set the learning rate through `eta0=0.01`.
- Since we are not using regularization, we set `alpha=0`.

```
estimator = SGDClassifier(loss='log',
                          penalty='l2',
                          max_iter=1,
                          warm_start=True,
                          eta0=0.01,
                          alpha=0,
                          learning_rate='constant',
                          random_state=1729)
pipe_sgd= make_pipeline(MinMaxScaler(), estimator)
```

- Let us call `fit` method of `SGDClassifier()` in iterative manner to plot the iteration vs loss curve.

```
Loss=[]
iterations= 100
for i in range(iterations):
  pipe_sgd.fit(x_train,y_train_0)
  y_pred = pipe_sgd.predict_proba(x_train)
  Loss.append(log_loss(y_train_0,y_pred))
```

Let's plot the learning curve.

- We have iterator number on the x-axis and loss on the y-axis.

```
plt.figure()
plt.plot(np.arange(iterations),Loss)
plt.grid(True)
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.show()
```

Observe that the loss is reducing iteration after iteration.

> It signals us the model is getting trained as per the expectation.

Now that the model is trained, let's us calculate the training and test accuracy of the model.

```
print('Training accuracy: %.2f'%pipe_sgd.score(x_train,y_train_0))
print('Testing accuracy: %.2f'%pipe_sgd.score(x_test,y_test_0))
```

- We know that accuracy alone is not a good metric for binary classification.
- Let's compute Precision, recall and f1-score for the model.

```
y_hat_train_0 = pipe_sgd.predict(x_train)
cm_display = ConfusionMatrixDisplay.from_predictions(y_train_0,y_hat_train_0,values_format
plt.show()
```

Observe that the off-diagonal elements are not zero, which indicates that a few examples are misclassified.

```
print(classification_report(y_train_0, y_hat_train_0, digits=3))
```

Let's generate classification report on test set.

```
y_hat_test_0 = pipe_sgd.predict(x_test)
cm_display = ConfusionMatrixDisplay.from_predictions(y_test_0,y_hat_test_0,values_format='
plt.show()
```

```
print(classification_report(y_test_0, y_hat_test_0, digits=3))
```

Cross validation

```
estimator = SGDClassifier(loss='log',
                          penalty='l2',
                          max_iter=100,
                          warm_start=False,
                          eta0=0.01,
                          alpha=0,
                          learning_rate='constant',
                          random_state=1729)

# create a pipeline
pipe_sgd_cv = make_pipeline(MinMaxScaler(),estimator)
```

Now we will train the classifier with cross validation.

```
cv_bin_clf = cross_validate(pipe_sgd_cv,
                            x_train,
                            y_train_0,
                            cv=5,
                            scoring=['precision','recall','f1'],
                            return_train_score=True,
                           )
pprint(cv_bin_clf)
```

- It is good to check the weight values of all the features which will help us decide whether regularization could be of any help.

```
# call the estimator object in the steps of pipeline
weights = pipe_sgd[1].coef_
bias = pipe_sgd[1].intercept_
print('Dimention of Weights w: {0}'.format(weights.shape))
print('Bias :{0}'.format(bias))


plt.figure()
plt.plot(np.arange(0,784),weights[0,:])
plt.xlabel('Feature index')
plt.ylabel('Weight value')
plt.ylim((np.min(weights)-5, np.max(weights)+5))
plt.grid()
```

- It is interesting to observe how many weight values are exactly zero.
- Those features contribute nothing in the classification.

```
num_zero_w = weights.shape[1]-np.count_nonzero(weights)
print('Number of weights with value zero:%f'%num_zero_w)
```

- Looking at the above plot and the performance of the model on training and test, it is obvious that the model do not require any regularization.

Training with regularization

- However, what happens to the performance of the model if we penalize, out of temptation, the weight values even to a smaller degree.
- Think about it.

[ ] ↳ *13 cells hidden*

Displaying input image and its prediction

[ ] ↳ *5 cells hidden*

Hyper-parameter tuning

- The learning rate and regularization rate are two important hyper-parameters of sgdclassifier.
- Let's use `RandomizedSearchCV()` and draw the value from the log-uniform distribution to find a better combination of `eta and alpha`

```
eta_grid = loguniform(1e-3,1e-1)
alpha_grid = loguniform(1e-7,1e-1)
```

- Note that, `eta_grid & alpha_grid` are objects that contain a method called `rvs()` which can be called to get random values of given size.
- Therefore, we pass this `eta_grid & alpha_grid` objects to `RandomizedSearchCV()`. Internally, it makes use of this `rvs()` method of these objects for sampling.

```
print(eta_grid.rvs(10, random_state=42))
```

```
estimator= SGDClassifier(loss='log',
                         penalty='l2',
                         max_iter=100,
                         warm_start=False,
                         learning_rate='constant',
                         eta0=0.01,
                         alpha=0,
                         random_state=1729)
```

```
pipe_sgd_hpt = make_pipeline(MinMaxScaler(),estimator)
print(pipe_sgd_hpt)


scores = RandomizedSearchCV(pipe_sgd_hpt,
                            param_distributions={
                                'sgdclassifier__eta0':eta_grid,
                                'sgdclassifier__alpha':alpha_grid
                            },
                            cv=5,
                            scoring='precision',
                            n_iter=10,
                            refit=True,
                            random_state=1729)


# It take quite a long time to finish
scores.fit(x_train,y_train_0)


pprint(scores.cv_results_)
```

- Let us pick the best estimator from the results

```
print('Best combination:(alpha:{0:.8f},eta:{1:.5f})'.format(scores.best_params_['sgdclassi


best_sgd_clf = scores.best_estimator_
y_hat_train_best_0 = best_sgd_clf.predict(x_train)


cm_display = ConfusionMatrixDisplay.from_predictions(y_train_0,y_hat_train_best_0,values_f
plt.show()


y_hat_test_best_0 = best_sgd_clf.predict(x_test)
cm_display = ConfusionMatrixDisplay.from_predictions(y_test_0,y_hat_test_best_0,values_for
plt.show()


print(classification_report(y_train_0,y_hat_train_best_0))


print(classification_report(y_test_0,y_hat_test_best_0))
```

Question: Why are the evaluation metric not better than `SGDClassifier` with manual parameter setting?

> We need to allow the hyper-parameter tuning process to run long enough. Try the procedure with more iterations - that will allow it to explore more in the parameter space and get us the best hyperparameters.

Learning Curve

[ ] ↳ *5 cells hidden*

Classification Report

Precision/Recall Tradeoff

[ ] ↳ *5 cells hidden*

Video#2

Logistic Regression

- In the previous setup, we used `SGDClassifier` to train 0-detector model in an iterative manner.
  - We can also train such a classifier by solving a set of equations obtained by setting the derivative of loss w.r.t. weights to 0.
  - These are not linear equations and therefore we need a different set of solvers.
- Sklearn uses solvers like `liblinear`, `newton-cg`, `sag`, `saga` and `lbfgs` to find the optimal weights.
- Regularization is applied by default.
- Parameters:

```
LogisticRegression(penalty='l2', *, dual=False, tol=0.0001, C=1.0,
fit_intercept=True, intercept_scaling=1, class_weight=None,
random_state=None, solver='lbfgs', max_iter=100,
multi_class='auto', verbose=0, warm_start=False, n_jobs=None,
l1_ratio=None)
```

- Note some of the important default parameters:
  - Regularization: `penalty='l2'``
  - Regularization rate: `C=1`
  - Solver: `solver = 'lbfgs'`
- Let's implement `LogisticRegression()` **without regularization** by setting the parameter $C = \infty$. Therefore, we may expect performance close to `SGDClassifier` without regularization

Training without regularization

- **STEP 1**: Instantiate a pipeline object with two stages:
  - The first stage contains `MinMaxScaler` for scaling the input.
  - The second state contains a `LogisticRegression` classifier with the regularization rate `C=infinity`.
- **STEP 2**: Train the pipeline with feature matrix `x_train` and label vector `y_train_0`.

```
pipe_logit = make_pipeline(MinMaxScaler(),LogisticRegression(random_state=1729,
                                                             solver='lbfgs',
                                                             C=np.infty))
pipe_logit.fit(x_train, y_train_0)
```

By executing this cell, we trained our `LogisticRegression` classifier, which can be used for making predictions on the new inputs.

Hyperparameter search

In this section, we will search for the best value for parameter `C` under certain `scoring` function.

with `GridSearchCV`

In the previous cell, we trained `LogisticRegression` classifier with `C=infinity`. You may wonder if that's the best value for `C` and if it is not the best value, how do we search for it?

Now we will demonstrate how to search for the best parameter value for regularization rate `C`, as an illustration, using `GridSearchCV`.

> Note that you can also use `RandomizedSearchCV` for this purpose.

In order to use `GridSearchCV`, we first define a set of values that we want to try out for `C`. The best value of `C` will be found from this set.

We define the `pipleline` object exactly like before with one exception: we have set parameter value C to 1 in `LogisticRegression` estimator. You can set it to any value as the best value would be searched with grid search.

The additional step here is to instatiate a `GridSearchCV` object with a `pipeline` estimator, parameter grid specification and `f1` as a `scoring` function.

> Note that you can use other scoring functions like `precision`, `recall`, however the value of `C` is found such that the given scoring function is optimized.

```
from sklearn.pipeline import Pipeline

grid_Cs = [0, 1e-4, 1e-3, 1e-2, 1e-1, 1.0, 10.0, 100.0]

scaler = MinMaxScaler()
logreg = LogisticRegression(C=1.0, random_state=1729)

pipe = Pipeline(steps=[("scaler", scaler),
                       ("logistic", logreg)])

pipe_logit_cv = GridSearchCV(
    pipe,
    param_grid={"logistic__C": grid_Cs},
    scoring='f1')
pipe_logit_cv.fit(x_train, y_train_0)
```

The `GridSearchCV` finds the best value of `C` and refits the estimator by default on the entire training set. This gives us the logistic regression classifier with best value of `C`.

We can check the value of the best parameter by accessing the `best_params_` member variable of the `GridSearchCV` object.

```
pipe_logit_cv.best_params_
```

and the best score can be found in `best_score_` member variable and can be obtained as follows:

```
pipe_logit_cv.best_score_
```

The best estimator can be accessed with `best_estimator_` member variable.

```
pipe_logit_cv.best_estimator_
```

With `LogisticRegressionCV`

Instead of using `GridSearchCV` for finding the best value for parameter `C`, we can use `LogisticRegressionCV` for performing the same job.

- **STEP 1**: Here we make use of `LogisticRegressionCV` estimator with number of cross validation folds `cv=5` and scoring scheme `scoring='f1'` in the `pipeline` object.
- **STEP 2**: In the second step, we train the pipeline object as before.

```
estimator = LogisticRegressionCV(cv=5, scoring='f1', random_state=1729)
logit_cv = make_pipeline(MinMaxScaler(), estimator)
```

```
logit_cv.fit(x_train, y_train_0)
```

By default, `LogisticRegressionCV` refits the model on the entire training set with the best parameter values obtained via cross validation.

Performance evaluation

Precision, recall and f1-score

Let's evaluate performance of these three different logistic regression classifiers for detecting digit 0 from the image.

- Logistic regression without regularization.
- Best logistic regression classifier found through `GridSearchCV`.
- Best classifier found through `LogisticRegressionCV`.

> Note that `GridSearchCV` and `LogisticRegressionCV` by default refit the classifier for the best hyperparameter values.

Let's get predicition for test set with these three classifiers:

```
lr_y_hat_0 = pipe_logit.predict(x_test)
lr_gs_y_hat_0 = pipe_logit_cv.best_estimator_.predict(x_test)
lr_cv_y_hat_0 = logit_cv.predict(x_test)
```

We will compare **precision**, **recall** and **F1 score** for the three classifiers.

```
precision_lr = precision_score(y_test_0, lr_y_hat_0)
recall_lr = recall_score(y_test_0, lr_y_hat_0)

precision_lr_gs = precision_score(y_test_0, lr_gs_y_hat_0)
recall_lr_gs = recall_score(y_test_0, lr_gs_y_hat_0)

precision_lr_cv = precision_score(y_test_0, lr_cv_y_hat_0)
recall_lr_cv = recall_score(y_test_0, lr_cv_y_hat_0)


print (f"LogReg: precision={precision_lr}, recall={recall_lr}")
print (f"GridSearch: precision={precision_lr_gs}, recall={recall_lr_gs}")
print (f"LogRegCV: precision={precision_lr_cv}, recall={recall_lr_cv}")
```

Note that all three classifiers have roughly the same performance as measured with precision and recall.

- The `LogisticRegression` classifier obtained through `GridSearchCV` has the highest precision - marginally higher than the other two classifiers.
- The `LogisticRegression` classifier obtained through `LogisticRegressionCV` has the highest recall - marginally higher than the other two classifiers.

Using PR-curve

```
y_scores_lr = pipe_logit.decision_function(x_test)
precisions_lr, recalls_lr, thresholds_lr = precision_recall_curve(
    y_test_0, y_scores_lr)

y_scores_lr_gs = pipe_logit_cv.decision_function(x_test)
precisions_lr_gs, recalls_lr_gs, thresholds_lr_gs = precision_recall_curve(
    y_test_0, y_scores_lr_gs)

y_scores_lr_cv = logit_cv.decision_function(x_test)
precisions_lr_cv, recalls_lr_cv, thresholds_lr_cv = precision_recall_curve(
    y_test_0, y_scores_lr_cv)
```

We have all the quantities for plotting the PR curve. Let's plot PR cuves for all three classifiers:

```
plt.figure(figsize=(10,4))
plt.plot(recalls_lr[:-1], precisions_lr[:-1], 'b--', label='LogReg')
plt.plot(recalls_lr_gs[:-1], precisions_lr_gs[:-1], 'r--', label='GridSearchCV')
plt.plot(recalls_lr_cv[:-1], precisions_lr_cv[:-1], 'g--', label='LogRegCV')

plt.ylabel('Precision')
plt.xlabel('Recall')
plt.grid(True)
plt.legend(loc='lower left')
plt.show()
```

Note that the PR curves for all three classifiers overlap significantly.

Let's calculate area under the PR curve:

```
from sklearn.metrics import auc

auc_lr = auc(recalls_lr[:-1], precisions_lr[:-1])
auc_lr_gs = auc(recalls_lr_gs[:-1], precisions_lr_gs[:-1])
auc_lr_cv = auc(recalls_lr_cv[:-1], precisions_lr_cv[:-1])


print ("AUC-PR for logistic regression:", auc_lr)
print ("AUC-PR for grid search:", auc_lr_gs)
print ("AUC-PR for logistic regression CV:", auc_lr_cv)
```

Observe that the AUC for all three classifier is roughly the same with `LogisticRegression` classifier obtained through cross validation and grid search have slightly higher AUC under PR curve.

Confusion matrix

We show a confusion matrix for test set with logistic regression classifier:

```
cm_display = ConfusionMatrixDisplay.from_predictions(y_test_0,lr_y_hat_0,
                                                     values_format='.5g')
# it return matplotlin plot object
plt.show()
```

Confusion matrix for test set with logistic regression classifier obtained through **grid search**:

```
cm_display = ConfusionMatrixDisplay.from_predictions(y_test_0,lr_gs_y_hat_0,
                                                     values_format='.5g')
# it return matplotlin plot object
plt.show()
```

Confusion matrix for test set with logistic regression classifier obtained through **cross validation**:

```
cm_display = ConfusionMatrixDisplay.from_predictions(y_test_0,lr_cv_y_hat_0,
                                                     values_format='.5g')
# it return matplotlin plot object
plt.show()
```

Exercise: Plot ROC curve for all three classifiers and calculate area under ROC curve.

Video #3

Ridge Classifier

- Ridge classifier cast the problem as least-square classification and finds the optimal weight using some matrix decomposition technique such as SVD.
- To train the ridge classifier, the labels should be $y \in \{+1, -1\}$.
- The classifier also by default implements L2 regularization. However, we first implement it without regularization by setting `alpha=0`

```
# intialize new variable names with all -1
y_train_0 = -1*np.ones((len(y_train)))
y_test_0 = -1*np.ones((len(y_test)))

# find indices of digit 0 image
indx_0 = np.where(y_train =='0')

# use those indices to modify y_train_0&y_test_0
y_train_0[indx_0] = 1
indx_0 = np.where(y_test == '0')
y_test_0[indx_0] = 1
```

- First take a look into the parameters of the class

> RidgeClassifier(alpha=1.0, *, fit_intercept=True, normalize='deprecated', copy_X=True, max_iter=None, tol=0.001, class_weight=None, solver='auto', positive=False, random_state=None)

- Note the parameter `"normalize"` is depreceated.

```
estimator = RidgeClassifier(normalize=False,alpha=0)
pipe_ridge = make_pipeline(MinMaxScaler(),estimator)
pipe_ridge.fit(x_train,y_train_0)
```

Performance

```
y_hat_test_0 = pipe_ridge.predict(x_test)
print(classification_report(y_test_0,y_hat_test_0))
```

Cross Validation

```
cv_bin_ridge_clf = cross_validate(
                      pipe_ridge, x_train, y_train_0, cv=5,
                      scoring=['precision', 'recall', 'f1'],
                      return_train_score=True,
                      return_estimator=True)
pprint(cv_bin_ridge_clf)
```

```
best_estimator_id = np.argmax(cv_bin_ridge_clf['train_f1']); best_estimator_id
```

```
best_estimator = cv_bin_ridge_clf['estimator'][best_estimator_id]
```

Let's evaluate the performance of the best classifier on the test set:

```
y_hat_test_0 = best_estimator.predict(x_test)
print(classification_report(y_test_0,y_hat_test_0))
```

Further exploration

Let's see what these classifiers leant about the digit 0

```
models = (pipe_sgd, pipe_sgd_l2, pipe_logit, pipe_ridge)
titles = ('sgd', 'regularized sgd', 'logit', 'ridge')
plt.figure(figsize=(4, 4))
plt.subplots(2, 2)
for i in range(0, 4):
  w = models[i][1].coef_
  w_matrix = w.reshape(28, 28)
  w_matrix[w_matrix < 0]=0 #just set the value less than zero to zero
  plt.subplot(2, 2, i+1)
  plt.imshow(w_matrix, cmap='gray')
  plt.title(titles[i])
  plt.axis('off')
  plt.grid(False)
fig.show()
```

Video 4

Multiclass Classifier (OneVsAll)

Multiclass Logit with SGD

```
estimator = SGDClassifier(loss='log',
                          penalty='l2',
                          max_iter=1,
                          warm_start=True,
                          eta0=0.01,
                          alpha=0,
                          learning_rate='constant',
                          random_state=1729)
pipe_sgd_ovr= make_pipeline(MinMaxScaler(),estimator)
```

It almost took 5 minutes for training.

```
Loss=[]
iterations= 100
for i in range(iterations):
```

```
  pipe_sgd_ovr.fit(x_train,y_train)
  y_pred = pipe_sgd_ovr.predict_proba(x_train)
  Loss.append(log_loss(y_train,y_pred))


plt.figure()
plt.plot(np.arange(iterations),Loss)
plt.grid(True)
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.show()
```

What actually happened behind the screen is that the library automatically created 10 binary classifiers and trained them!. During the inference time, the input will be passed through all the 10 classifiers and the highest score among the ouputs will be considered as the predicted class.To see it in action, let us execute the following lines of code

```
pipe_sgd_ovr[1]
```

```
pipe_sgd_ovr[1].coef_.shape
```

So it is a matrix of size 10×784. A row represents the weights of a single binary classifier.

```
y_hat = pipe_sgd_ovr.predict(x_test); y_hat[:5]
```

```
cm_display = ConfusionMatrixDisplay.from_predictions(y_test, y_hat,
                                                     values_format='.5g')
plt.show()
```

```
print(classification_report(y_test, y_hat))
```

Multi-class LogisticRegression using solvers

```
pipe_logit_ovr = make_pipeline(MinMaxScaler(),
                               LogisticRegression(random_state=1729,
                                                  solver='lbfgs',
                                                  C=np.infty))
pipe_logit_ovr.fit(x_train,y_train)
```

```
y_hat = pipe_logit_ovr.predict(x_test)
cm_display = ConfusionMatrixDisplay.from_predictions(y_test, y_hat,
                                                     values_format='.5g')
plt.show()
```

```
print(classification_report(y_test, y_hat))
```

Visualize the weight values

```
W = pipe_logit_ovr[1].coef_
# normalize
W = MinMaxScaler().fit_transform(W)
fig,ax = plt.subplots(3,3)
index = 1
for i in range(3):
  for j in range(3):
    ax[i][j].imshow(W[index,:].reshape(28,28),cmap='gray')
    ax[i][j].set_title('W{0}'.format(index))
    ax[i][j].set_axis_off()
    index += 1
```

Excersise: Multiclass classification with RidgeClassifier.