Module 5

Date	@30/08/2023
Title	Linear Classifier in Python

Regularized logistic regression

In Chapter 1, you used logistic regression on the handwritten digits data set. Here, we'll explore the effect of L2 regularization.

The handwritten digits dataset is already loaded, split, and stored in the variables `X_train`, `y_train`, `X_valid`, and `y_valid`. The variables `train_errs` and `valid_errs` are already initialized as empty lists.

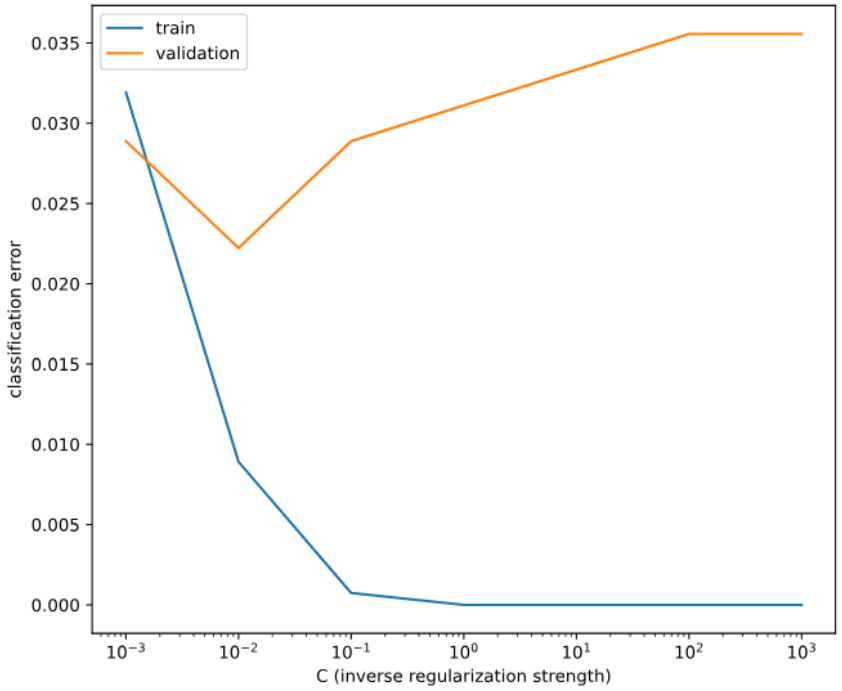
- Loop over the different values of `C_value`, creating and fitting a `LogisticRegression` model each time.
- Save the error on the training set and the validation set for each model.
- Create a plot of the training and testing error as a function of the regularization parameter, `C`.
- Looking at the plot, what's the best value of `C`?

```
# Train and validaton errors initialized as empty list
train_errs = list()
valid_errs = list()

# Loop over values of C_value
for C_value in [0.001, 0.01, 0.1, 1, 10, 100, 1000]:
    # Create LogisticRegression object and fit
    lr = LogisticRegression(penalty='l2', C=C_value)
    lr.fit(X_train,y_train)

    # Evaluate error rates and append to lists
    train_errs.append( 1.0 - lr.score(X_train, y_train))
    valid_errs.append( 1.0 - lr.score(X_valid, y_valid))
```

```
# Plot results
plt.semilogx(C_values, train_errs, C_values, valid_errs)
plt.legend(("train", "validation"))
plt.show()
```



Logistic regression and feature selection

In this exercise we'll perform feature selection on the movie review sentiment data set using L1 regularization. The features and targets are already loaded for you in `X_train` and `y_train`.

We'll search for the best value of `C` using scikit-learn's `GridSearchCV()`, which was covered in the prerequisite course.

Instructions70 XP

- Instantiate a logistic regression object that uses L1 regularization.
- Find the value of `C` that minimizes cross-validation error.
- Print out the number of selected features for this value of `C`.

```
# Specify L1 regularization
lr = LogisticRegression(solver='liblinear', penalty='l1')

# Instantiate the GridSearchCV object and run the search
searcher = GridSearchCV(lr, {'C':[0.001, 0.01, 0.1, 1, 10]})
searcher.fit(X_train, y_train)

# Report the best parameters
print("Best CV params", searcher.best_params_)

# Find the number of nonzero coefficients (selected features)
best_lr = searcher.best_estimator_
coefs = best_lr.coef_
print("Total number of features:", coefs.size)
print("Number of selected features:", np.count_nonzero(coefs)
```

```
Best CV params {'C': 1}
Total number of features: 2500
Number of selected features: 1219
```

Identifying the most positive and negative words

In this exercise we'll try to interpret the coefficients of a logistic regression fit on the movie review sentiment dataset. The model object is already instantiated and fit for you in the variable `lr`.

In addition, the words corresponding to the different features are loaded into the variable `vocab`. For example, since `vocab[100]` is "think", that means feature 100 corresponds to the number of times the word "think" appeared in that movie review.

- Find the words corresponding to the 5 largest coefficients.
- Find the words corresponding to the 5 smallest coefficients.

```
# Get the indices of the sorted cofficients
inds_ascending = np.argsort(lr.coef_.flatten())
inds_descending = inds_ascending[::-1]
```

```
# Print the most positive words
print("Most positive words: ", end="")
for i in range(5):
    print(vocab[inds_descending[i]], end=", ")
print("\n")

# Print most negative words
print("Most negative words: ", end="")
for i in range(5):
    print(vocab[inds_ascending[i]], end=", ")
print("\n")
```

```
Most positive words: favorite, superb, noir, knowing, excellent,

Most negative words: worst, disappointing, waste, boring, lame,
```

Fitting multi-class logistic regression

In this exercise, you'll fit the two types of multi-class logistic regression, one-vs-rest and softmax/multinomial, on the handwritten digits data set and compare the results. The handwritten digits dataset is already loaded and split into `X_train`, `y_train`, `X_test`, and `y_test`.

- Fit a one-vs-rest logistic regression classifier by setting the `multi_class` parameter and report the results.
- Fit a multinomial logistic regression classifier by setting the `multi_class` parameter and report the results.

```
# Fit one-vs-rest logistic regression classifier
lr_ovr = LogisticRegression(multi_class='ovr')
lr_ovr.fit(X_train, y_train)

print("OVR training accuracy:", lr_ovr.score(X_train, y_train
print("OVR test accuracy    :", lr_ovr.score(X_test, y_test))

# Fit softmax classifier
```

```
lr_mn = LogisticRegression(multi_class='multinomial', solver=
lr_mn.fit(X_train, y_train)

print("Softmax training accuracy:", lr_mn.score(X_train, y_tr
print("Softmax test accuracy    :", lr_mn.score(X_test, y_tes
```

```
OVR training accuracy: 0.9955456570155902
OVR test accuracy    : 0.9644444444444444
Softmax training accuracy: 1.0
Softmax test accuracy    : 0.9688888888888889
```

Visualizing multi-class logistic regression

In this exercise we'll continue with the two types of multi-class logistic regression, but on a toy 2D data set specifically designed to break the one-vs-rest scheme.

The data set is loaded into `X_train` and `y_train`. The two logistic regression objects, `lr_mn` and `lr_ovr`, are already instantiated (with `C=100`), fit, and plotted.

Notice that `lr_ovr` never predicts the dark blue class... yikes! Let's explore why this happens by plotting one of the binary classifiers that it's using behind the scenes.

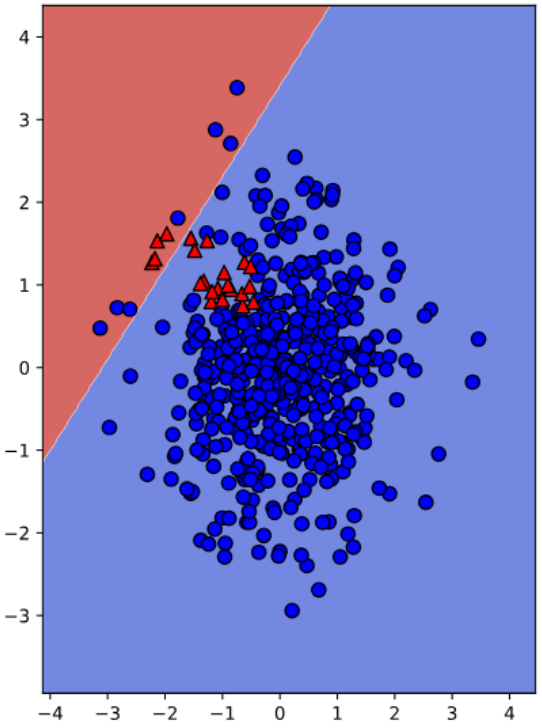
- Create a new logistic regression object (also with `C=100`) to be used for binary classification.
- Visualize this binary classifier with `plot_classifier`... does it look reasonable?

```
# Print training accuracies
print("Softmax     training accuracy:", lr_mn.score(X_train,
print("One-vs-rest training accuracy:", lr_ovr.score(X_train,

# Create the binary classifier (class 1 vs. rest)
lr_class_1 = LogisticRegression(C=100)
lr_class_1.fit(X_train, y_train==1)

# Plot the binary classifier (class 1 vs. rest)
plot_classifier(X_train, y_train==1, clf=lr_class_1)
```

```
Softmax     training accuracy: 0.996
One-vs-rest training accuracy: 0.916
```



One-vs-rest SVM

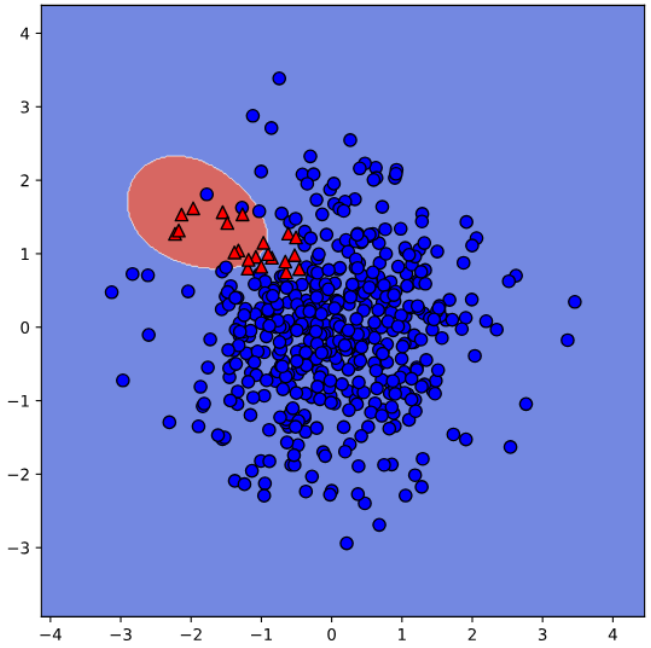
As motivation for the next and final chapter on support vector machines, we'll repeat the previous exercise with a non-linear SVM. Once again, the data is loaded into `X_train`, `y_train`, `X_test`, and `y_test`.

Instead of using `LinearSVC`, we'll now use scikit-learn's `SVC` object, which is a non-linear "kernel" SVM (much more on what this means in Chapter 4!). Again, your task is to create a plot of the binary classifier for class 1 vs. rest.

- Fit an `SVC` called `svm_class_1` to predict class 1 vs. other classes.
- Plot this classifier.

```
# We'll use SVC instead of LinearSVC from now on
from sklearn.svm import SVC

# Create/plot the binary classifier (class 1 vs. rest)
svm_class_1 = SVC()
svm_class_1.fit(X_train, y_train == 1)
plot_classifier(X_train, y_train == 1, svm_class_1)
```



Effect of removing examples

Support vectors are defined as training examples that influence the decision boundary. In this exercise, you'll observe this behavior by removing non support vectors from the training set.

The wine quality dataset is already loaded into `X` and `y` (first two features only). (Note: we specify `lims` in `plot_classifier()` so that the two plots are forced to use the same axis limits and can be compared directly.)

- Train a linear SVM on the whole data set.
- Create a new data set containing only the support vectors.
- Train a new linear SVM on the smaller data set.

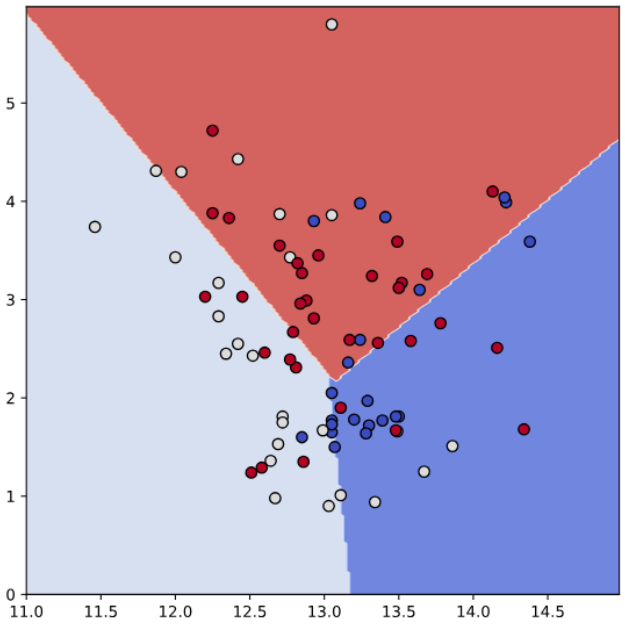
```
# Train a linear SVM
svm = SVC(kernel="linear")
svm.fit(X, y)
plot_classifier(X, y, svm, lims=(11,15,0,6))

# Make a new data set keeping only the support vectors
print("Number of original examples", len(X))
print("Number of support vectors", len(svm.support_))
X_small = X[svm.support_]
```

```
y_small = y[svm.support_]

# Train a new SVM using only the support vectors
svm_small = SVC(kernel="linear")
svm_small.fit(X_small, y_small)
plot_classifier(X_small, y_small, svm_small, lims=(11,15,0,6)
```

```
Number of original examples 178
Number of support vectors 81
```



GridSearchCV warm-up

In the video we saw that increasing the RBF kernel hyperparameter `gamma` increases training accuracy. In this exercise we'll search for the `gamma` that maximizes cross-validation accuracy using scikit-learn's `GridSearchCV`. A binary version of the handwritten digits dataset, in which you're just trying to predict whether or not an image is a "2", is already loaded into the variables `X` and `y`.

- Create a `GridSearchCV` object.
- Call the `fit()` method to select the best value of `gamma` based on cross-validation accuracy.

```
# Instantiate an RBF SVM
svm = SVC()
```

```
# Instantiate the GridSearchCV object and run the search
parameters = {'gamma':[0.00001, 0.0001, 0.001, 0.01, 0.1]}
searcher = GridSearchCV(svm, parameters, cv=5)
searcher.fit(X, y)

# Report the best parameters
print("Best CV params", searcher.best_params_)
```

Jointly tuning gamma and C with GridSearchCV

In the previous exercise the best value of `gamma` was 0.001 using the default value of `C`, which is 1. In this exercise you'll search for the best combination of `C` and `gamma` using `GridSearchCV`.

As in the previous exercise, the 2-vs-not-2 digits dataset is already loaded, but this time it's split into the variables `X_train`, `y_train`, `X_test`, and `y_test`. Even though cross-validation already splits the training set into parts, it's often a good idea to hold out a separate test set to make sure the cross-validation results are sensible.

- Run `GridSearchCV` to find the best hyperparameters using the training set.
- Print the best values of the parameters.
- Print out the accuracy on the test set, which was not used during the cross-validation procedure.

```
# Instantiate an RBF SVM
svm = SVC()

# Instantiate the GridSearchCV object and run the search
parameters = {'C':[0.1, 1, 10], 'gamma':[0.00001, 0.0001, 0.0
searcher = GridSearchCV(svm, parameters, cv=5)
searcher.fit(X_train, y_train)

# Report the best parameters and the corresponding score
print("Best CV params", searcher.best_params_)
print("Best CV accuracy", searcher.best_score_)
```

```
# Report the test accuracy using these best parameters
print("Test accuracy of best grid search hypers:", searcher.s
```

```
Best CV params {'C': 1, 'gamma': 0.001}
Best CV accuracy 0.9988826815642458
Test accuracy of best grid search hypers: 0.9988876529477196
```

Using SGDClassifier

In this final coding exercise, you'll do a hyperparameter search over the regularization strength and the loss (logistic regression vs. linear SVM) using `SGDClassifier()`

- Instantiate an `SGDClassifier` instance with `random_state=0`.
- Search over the regularization strength and the `hinge` vs. `log_loss` losses.

```
# We set random_state=0 for reproducibility
linear_classifier = SGDClassifier(random_state=0)

# Instantiate the GridSearchCV object and run the search
parameters = {'alpha':[0.00001, 0.0001, 0.001, 0.01, 0.1, 1],
              'loss':['hinge','log_loss']}
searcher = GridSearchCV(linear_classifier, parameters, cv=10)
searcher.fit(X_train, y_train)

# Report the best parameters and the corresponding score
print("Best CV params", searcher.best_params_)
print("Best CV accuracy", searcher.best_score_)
print("Test accuracy of best grid search hypers:", searcher.s
```

```
Best CV params {'alpha': 0.001, 'loss': 'hinge'}
Best CV accuracy 0.9490730158730158
Test accuracy of best grid search hypers: 0.9611111111111111
```