1. What is the underlying concept of Support Vector Machines ?

Ans : The fundamental idea behind Support Vector Machines is to fit the widest possible “street” between the classes. In other words, the goal is to have the largest possible margin between the decision boundary that separates the two classes and the training instances. When performing soft margin classification, the SVM searches for a compromise between perfectly separating the two classes and having the widest possible street (i.e., a few instances may end up on the street). Another key idea is to use kernels when training on nonlinear datasets.

2. What is the concept of a support vector ?

Ans : After training an SVM, a support vector is any instance located on the “street” (see the previous answer), including its border. The decision boundary is entirely determined by the support vectors. Any instance that is not a support vector (i.e., off the street) has no influence whatsoever; you could remove them, add more instances, or move them around, and as long as they stay off the street they won’t affect the decision boundary. Computing the predictions only involves the support vectors, not the whole training set.

3. When using SVMs, why is it necessary to scale the inputs ?

Ans : SVMs try to fit the largest possible “street” between the classes (see the first answer), so if the training set is not scaled, the SVM will tend to neglect small features.

4. When an SVM classifier classifies a case, can it output a confidence score? What about a percentage chance ?

Ans : An SVM classifier can output the distance between the test instance and the decision boundary, and you can use this as a confidence score. However, this score cannot be directly converted into an estimation of the class probability. If you set probability=True when creating an SVM in Scikit-Learn, then after training it will calibrate the probabilities using Logistic Regression on the SVM’s scores (trained by an additional five-fold cross-validation on the training data). This will add the predict\_proba() and predict\_log\_proba() methods to the SVM.

5. Should you train a model on a training set with millions of instances and hundreds of features using the primal or dual form of the SVM problem ?

Ans : This question applies only to linear SVMs since kernelized can only use the dual form. The computational complexity of the primal form of the SVM problem is proportional to the number of training instances m, while the computational complexity of the dual form is proportional to a number between m2 and m3. So if there are millions of instances, you should definitely use the primal form, because the dual form will be much too slow.

6. Let's say you've used an RBF kernel to train an SVM classifier, but it appears to underfit the training collection. Is it better to raise or lower (gamma)? What about the letter C ?

Ans : If an SVM classifier trained with an RBF kernel underfits the training set, there might be too much regularization. To decrease it, you need to increase gamma or C (or both).

7. To solve the soft margin linear SVM classifier problem with an off-the-shelf QP solver, how should the QP parameters (H, f, A, and b) be set ?

Ans : Let’s call the QP parameters for the hard-margin problem H′, f′, A′ and b′. The QP parameters for the soft-margin problem have m additional parameters (np = n + 1 + m) and m additional constraints (nc = 2m). They can be defined like so:

H is equal to H′, plus m columns of 0s on the right and m rows of 0s at the bottom

f is equal to f′ with m additional elements, all equal to the value of the hyperparameter C.

b is equal to b′ with m additional elements, all equal to 0.

A is equal to A′, with an extra m × m identity matrix Im appended to the right,

8. On a linearly separable dataset, train a LinearSVC. Then, using the same dataset, train an SVC and an SGDClassifier. See if you can get them to make a model that is similar to yours ?

Ans : it depends on the dataset and of course a lot of other factors add weight but today in this small post I’ll demonstrate how to use LinearSVC , SVC classifier and SGD classifier via Python code and also compare results for the same dataset.

Since classifiers are very sensitive to outliers we need to scale them but before we need to pick the right dataset, for simpler results I’ll showcase the benchmark dataset for all classifiers

Let's use the Iris dataset: the Iris Setosa and Iris Versicolor classes are linearly separable.

from sklearn import datasets

iris = datasets.load\_iris()

X = iris["data"][:, (2, 3)] # petal length, petal width

y = iris["target"]

setosa\_or\_versicolor = (y == 0) | (y == 1)

X = X[setosa\_or\_versicolor]

y = y[setosa\_or\_versicolor]

from sklearn.svm import SVC, LinearSVC

from sklearn.linear\_model import SGDClassifier

from sklearn.preprocessing import StandardScaler

C = 5

alpha = 1 / (C \* len(X))

lin\_clf = LinearSVC(loss="hinge", C=C, random\_state=42)

svm\_clf = SVC(kernel="linear", C=C)

sgd\_clf = SGDClassifier(loss="hinge", learning\_rate="constant", eta0=0.001, alpha=alpha,

max\_iter=1000, tol=1e-3, random\_state=42)

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

lin\_clf.fit(X\_scaled, y)

svm\_clf.fit(X\_scaled, y)

sgd\_clf.fit(X\_scaled, y)

print("LinearSVC: ", lin\_clf.intercept\_, lin\_clf.coef\_)

print("SVC: ", svm\_clf.intercept\_, svm\_clf.coef\_)

print("SGDClassifier(alpha={:.5f}):".format(sgd\_clf.alpha), sgd\_clf.intercept\_, sgd\_clf.coef\_)

LinearSVC: [0.28475098] [[1.05364854 1.09903804]]

SVC: [0.31896852] [[1.1203284 1.02625193]]

SGDClassifier(alpha=0.00200): [0.117] [[0.77714169 0.72981762]]

Let's plot the decision boundaries of these three models:

# Compute the slope and bias of each decision boundary

w1 = -lin\_clf.coef\_[0, 0]/lin\_clf.coef\_[0, 1]

b1 = -lin\_clf.intercept\_[0]/lin\_clf.coef\_[0, 1]

w2 = -svm\_clf.coef\_[0, 0]/svm\_clf.coef\_[0, 1]

b2 = -svm\_clf.intercept\_[0]/svm\_clf.coef\_[0, 1]

w3 = -sgd\_clf.coef\_[0, 0]/sgd\_clf.coef\_[0, 1]

b3 = -sgd\_clf.intercept\_[0]/sgd\_clf.coef\_[0, 1]

# Transform the decision boundary lines back to the original scale

line1 = scaler.inverse\_transform([[-10, -10 \* w1 + b1], [10, 10 \* w1 + b1]])

line2 = scaler.inverse\_transform([[-10, -10 \* w2 + b2], [10, 10 \* w2 + b2]])

line3 = scaler.inverse\_transform([[-10, -10 \* w3 + b3], [10, 10 \* w3 + b3]])

# Plot all three decision boundaries

plt.figure(figsize=(11, 4))

plt.plot(line1[:, 0], line1[:, 1], "k:", label="LinearSVC")

plt.plot(line2[:, 0], line2[:, 1], "b--", linewidth=2, label="SVC")

plt.plot(line3[:, 0], line3[:, 1], "r-", label="SGDClassifier")

plt.plot(X[:, 0][y==1], X[:, 1][y==1], "bs") # label="Iris versicolor"

plt.plot(X[:, 0][y==0], X[:, 1][y==0], "yo") # label="Iris setosa"

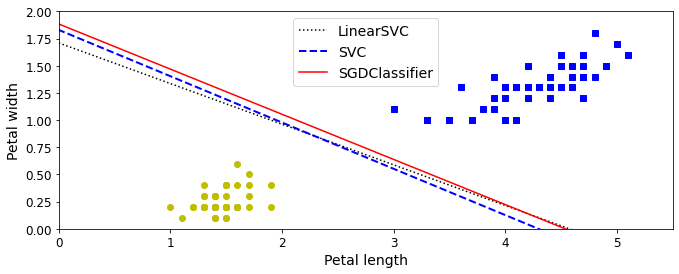
plt.xlabel("Petal length", fontsize=14)

plt.ylabel("Petal width", fontsize=14)

plt.legend(loc="upper center", fontsize=14)

plt.axis([0, 5.5, 0, 2])

plt.show().



9. On the MNIST dataset, train an SVM classifier. You'll need to use one-versus-the-rest to assign all 10 digits because SVM classifiers are binary classifiers. To accelerate up the process, you might want to tune the hyperparameters using small validation sets. What level of precision can you achieve ?

Ans : First, let's load the dataset and split it into a training set and a test set. We could use train\_test\_split() but people usually just take the first 60,000 instances for the training set, and the last 10,000 instances for the test set (this makes it possible to compare your model's performance with others):

Warning: since Scikit-Learn 0.24, fetch\_openml() returns a Pandas DataFrame by default. To avoid this, we use as\_frame=False.

from sklearn.datasets import fetch\_openml

mnist = fetch\_openml('mnist\_784', version=1, cache=True, as\_frame=False)

X = mnist["data"]

y = mnist["target"].astype(np.uint8)

X\_train = X[:60000]

y\_train = y[:60000]

X\_test = X[60000:]

y\_test = y[60000:]

Many training algorithms are sensitive to the order of the training instances, so it's generally good practice to shuffle them first. However, the dataset is already shuffled, so we do not need to do it.

Let's start simple, with a linear SVM classifier. It will automatically use the One-vs-All (also called One-vs-the-Rest, OvR) strategy, so there's nothing special we need to do. Easy!

Warning: this may take a few minutes depending on your hardware.

lin\_clf = LinearSVC(random\_state=42)

lin\_clf.fit(X\_train, y\_train)

/Users/ageron/miniconda3/envs/tf2/lib/python3.7/site-packages/sklearn/svm/\_base.py:977: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.

"the number of iterations.", ConvergenceWarning)

LinearSVC(random\_state=42)

Let's make predictions on the training set and measure the accuracy (we don't want to measure it on the test set yet, since we have not selected and trained the final model yet):

from sklearn.metrics import accuracy\_score

y\_pred = lin\_clf.predict(X\_train)

accuracy\_score(y\_train, y\_pred)

0.8348666666666666

Okay, 83.5% accuracy on MNIST is pretty bad. This linear model is certainly too simple for MNIST, but perhaps we just needed to scale the data first:

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train.astype(np.float32))

X\_test\_scaled = scaler.transform(X\_test.astype(np.float32))

Warning: this may take a few minutes depending on your hardware.

lin\_clf = LinearSVC(random\_state=42)

lin\_clf.fit(X\_train\_scaled, y\_train)

/Users/ageron/miniconda3/envs/tf2/lib/python3.7/site-packages/sklearn/svm/\_base.py:977: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.

"the number of iterations.", ConvergenceWarning)

LinearSVC(random\_state=42)

y\_pred = lin\_clf.predict(X\_train\_scaled)

accuracy\_score(y\_train, y\_pred)

0.9217333333333333

That's much better (we cut the error rate by about 53%), but still not great at all for MNIST. If we want to use an SVM, we will have to use a kernel. Let's try an SVC with an RBF kernel (the default).

Note: to be future-proof we set gamma="scale" since it will be the default value in Scikit-Learn 0.22.

svm\_clf = SVC(gamma="scale")

svm\_clf.fit(X\_train\_scaled[:10000], y\_train[:10000])

SVC()

y\_pred = svm\_clf.predict(X\_train\_scaled)

accuracy\_score(y\_train, y\_pred)

0.9455333333333333

That's promising, we get better performance even though we trained the model on 6 times less data. Let's tune the hyperparameters by doing a randomized search with cross validation. We will do this on a small dataset just to speed up the process:

from sklearn.model\_selection import RandomizedSearchCV

from scipy.stats import reciprocal, uniform

param\_distributions = {"gamma": reciprocal(0.001, 0.1), "C": uniform(1, 10)}

rnd\_search\_cv = RandomizedSearchCV(svm\_clf, param\_distributions, n\_iter=10, verbose=2, cv=3)

rnd\_search\_cv.fit(X\_train\_scaled[:1000], y\_train[:1000])

Fitting 3 folds for each of 10 candidates, totalling 30 fits

[CV] C=5.847490967837556, gamma=0.004375955271336425 .................

[Parallel(n\_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.

[CV] .. C=5.847490967837556, gamma=0.004375955271336425, total= 0.8s

[CV] C=5.847490967837556, gamma=0.004375955271336425 .................

[Parallel(n\_jobs=1)]: Done 1 out of 1 | elapsed: 0.8s remaining: 0.0s

[CV] .. C=5.847490967837556, gamma=0.004375955271336425, total= 0.8s

[CV] C=5.847490967837556, gamma=0.004375955271336425 .................

[CV] .. C=5.847490967837556, gamma=0.004375955271336425, total= 0.8s

[CV] C=2.544266730893301, gamma=0.024987648190235304 .................

[CV] .. C=2.544266730893301, gamma=0.024987648190235304, total= 0.9s

[CV] C=2.544266730893301, gamma=0.024987648190235304 .................

[CV] .. C=2.544266730893301, gamma=0.024987648190235304, total= 0.9s

[CV] C=2.544266730893301, gamma=0.024987648190235304 .................

[CV] .. C=2.544266730893301, gamma=0.024987648190235304, total= 0.9s

rnd\_search\_cv.best\_estimator\_

SVC(C=3.8786881587000437, gamma=0.0017076019229344522)

rnd\_search\_cv.best\_score\_

0.8599947252641863

This looks pretty low but remember we only trained the model on 1,000 instances. Let's retrain the best estimator on the whole training set:

Warning: the following cell may take hours to run, depending on your hardware.

rnd\_search\_cv.best\_estimator\_.fit(X\_train\_scaled, y\_train)

SVC(C=3.8786881587000437, gamma=0.0017076019229344522)

y\_pred = rnd\_search\_cv.best\_estimator\_.predict(X\_train\_scaled)

accuracy\_score(y\_train, y\_pred)

0.9978166666666667

Ah, this looks good! Let's select this model. Now we can test it on the test set:

y\_pred = rnd\_search\_cv.best\_estimator\_.predict(X\_test\_scaled)

accuracy\_score(y\_test, y\_pred)

0.9717

Not too bad, but apparently the model is overfitting slightly. It's tempting to tweak the hyperparameters a bit more (e.g. decreasing C and/or gamma), but we would run the risk of overfitting the test set. Other people have found that the hyperparameters C=5 and gamma=0.005 yield even better performance (over 98% accuracy). By running the randomized search for longer and on a larger part of the training set, you may be able to find this as well.

10. On the California housing dataset, train an SVM regressor ?

Ans : Let's load the dataset using Scikit-Learn's fetch\_california\_housing() function:

from sklearn.datasets import fetch\_california\_housing

housing = fetch\_california\_housing()

X = housing["data"]

y = housing["target"]

Split it into a training set and a test set:

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

Don't forget to scale the data:

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

Let's train a simple LinearSVR first:

from sklearn.svm import LinearSVR

lin\_svr = LinearSVR(random\_state=42)

lin\_svr.fit(X\_train\_scaled, y\_train)

/Users/ageron/miniconda3/envs/tf2/lib/python3.7/site-packages/sklearn/svm/\_base.py:977: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.

"the number of iterations.", ConvergenceWarning)

LinearSVR(random\_state=42)

Let's see how it performs on the training set:

from sklearn.metrics import mean\_squared\_error

y\_pred = lin\_svr.predict(X\_train\_scaled)

mse = mean\_squared\_error(y\_train, y\_pred)

mse

0.9641780189948642

Let's look at the RMSE:

np.sqrt(mse)

0.9819256687727764

In this training set, the targets are tens of thousands of dollars. The RMSE gives a rough idea of the kind of error you should expect (with a higher weight for large errors): so with this model we can expect errors somewhere around $10,000. Not great. Let's see if we can do better with an RBF Kernel. We will use randomized search with cross validation to find the appropriate hyperparameter values for C and gamma:

from sklearn.svm import SVR

from sklearn.model\_selection import RandomizedSearchCV

from scipy.stats import reciprocal, uniform

param\_distributions = {"gamma": reciprocal(0.001, 0.1), "C": uniform(1, 10)}

rnd\_search\_cv = RandomizedSearchCV(SVR(), param\_distributions, n\_iter=10, verbose=2, cv=3, random\_state=42)

rnd\_search\_cv.fit(X\_train\_scaled, y\_train)

[Parallel(n\_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.

Fitting 3 folds for each of 10 candidates, totalling 30 fits

[CV] C=4.745401188473625, gamma=0.07969454818643928 ..................

[CV] ... C=4.745401188473625, gamma=0.07969454818643928, total= 4.7s

[CV] C=4.745401188473625, gamma=0.07969454818643928 ..................

[Parallel(n\_jobs=1)]: Done 1 out of 1 | elapsed: 4.7s remaining: 0.0s

[CV] ... C=4.745401188473625, gamma=0.07969454818643928, total= 4.6s

[CV] C=4.745401188473625, gamma=0.07969454818643928 ..................

[CV] ... C=4.745401188473625, gamma=0.07969454818643928, total= 4.7s

[CV] C=8.31993941811405, gamma=0.015751320499779724 ..................

[CV] ... C=8.31993941811405, gamma=0.015751320499779724, total= 4.3s

[CV] C=8.31993941811405, gamma=0.015751320499779724 ..................

[CV] ... C=8.31993941811405, gamma=0.015751320499779724, total= 4.2s

rnd\_search\_cv.best\_estimator\_

SVR(C=4.745401188473625, gamma=0.07969454818643928)

Now let's measure the RMSE on the training set:

y\_pred = rnd\_search\_cv.best\_estimator\_.predict(X\_train\_scaled)

mse = mean\_squared\_error(y\_train, y\_pred)

np.sqrt(mse)

0.5727524770785359

Looks much better than the linear model. Let's select this model and evaluate it on the test set:

y\_pred = rnd\_search\_cv.best\_estimator\_.predict(X\_test\_scaled)

mse = mean\_squared\_error(y\_test, y\_pred)

np.sqrt(mse)

0.5929168385528734.