COMP5318 Week 6: Support Vector Machines. Dimensionality Reduction.

1. Setup

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os
from scipy import signal

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.pipeline import Pipeline

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

# For accuracy_score, classification_report and confusion_matrix
from sklearn import metrics
from sklearn.metrics import accuracy_score

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)

# Where to save the figures
PROJECT_ROOT_DIR = "."
TOPIC_ID = "svm"
IMAGES_PATH = os.path.join(PROJECT_ROOT_DIR, "images", TOPIC_ID)
os.makedirs(IMAGES_PATH, exist_ok=True)

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
    path = os.path.join(IMAGES_PATH, fig_id + "." + fig_extension)
    print("Saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)

# For plotting MNIST digits in the last exercise
def plot_digits(instances, images_per_row=5, **options):
    size = 28
    images_per_row = min(len(instances), images_per_row)
    images = [instance.reshape(size,size) for instance in instances]
    n_rows = (len(instances) - 1) // images_per_row + 1
    row_images = []
    n_empty = n_rows * images_per_row - len(instances)
    images.append(np.zeros((size, size * n_empty)))
    for row in range(n_rows):
        rimages = images[row * images_per_row : (row + 1) * images_per_row]
        row_images.append(np.concatenate(rimages, axis=1))
    image = np.concatenate(row_images, axis=0)
    plt.imshow(image, cmap = mpl.cm.binary, **options)
    plt.axis("off")
```

2. Support Vector Machines - Introduction

Support Vector Machines (SVMs) are powerful machine learning methods, able to form both linear and non-linear decision boundaries. They can be used for both classification and regression. In this tutorial we will learn how to implement linear and nonlinear SVMs for classification using **sklearn**. We will also explore the effect of the parameters on SVM's performance.

3. Prepare and load the data

We will use the breast cancer datset as an example. SVM classifiers work better when the features are on the same scale, so we will normalise the data:

In [2]:
```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()

# Create the training and test sets

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=42)

# Normalise data
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()#creating an object
scaler.fit(X_train)#calculate min and max value of the training data
X_train_norm = scaler.transform(X_train) #apply normalisation to the training set
X_test_norm = scaler.transform(X_test) #apply normalization to the test set
```

4. Create SVM classifiers

We wil use the SVC class to create linear and non-linear SVM classifiers. SVC stands for "Support Vector Classifier".

We start with creating a linear SVM. We need to set the kernel parametet to "linear"; the default is "rbf", corresponding to Radial-Basis Function (RBF) kernel, i.e. a non-linear SVM. We can see the other default parameters:

In [3]:
```
from sklearn.svm import SVC
lin_svm = SVC(kernel="linear")
lin_svm.fit(X_train_norm, y_train)
```

Out[3]:
```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
    kernel='linear', max_iter=-1, probability=False, random_state=None,
    shrinking=True, tol=0.001, verbose=False)
```

Testing the linear SVM on the test data:

In [4]:
```
y_pred = lin_svm.predict(X_test_norm)
print("Linear SVM - accuracy on test set: {:.3f}".format(accuracy_score(y_test, y_pred)))
```

```
Linear SVM - accuracy on test set: 0.979
```

Now let's create two other SVMs - nonlinear, with polynomial and RBF kernels respectively, and compare the results.

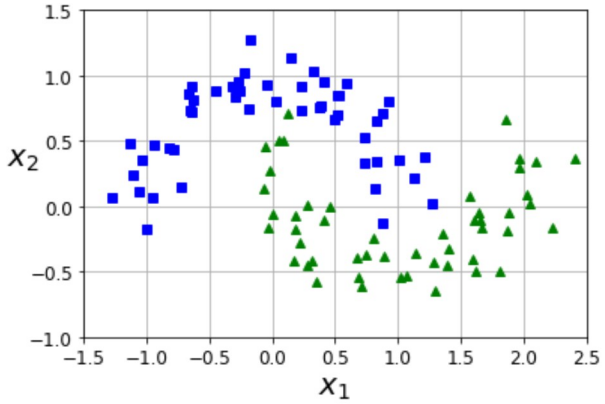
In [5]:
```
# SVM with polynomial kernel
poly_svm = SVC(kernel="poly", degree=2)#polynomial kernel with degree 2
poly_svm.fit(X_train_norm, y_train)
y_pred = poly_svm.predict(X_test_norm)
print("SVM with polynomial kernel - accuracy on test set: {:.3f}".format(accuracy_score(y_test, y_pred)))

# SVM with RBF kernel
rbf_svm = SVC(kernel="rbf", gamma="auto")
rbf_svm.fit(X_train_norm, y_train)
y_pred = rbf_svm.predict(X_test_norm)
print("SVM with RBF kernel - accuracy on test set: {:.3f}".format(accuracy_score(y_test, y_pred)))
```

```
SVM with polynomial kernel - accuracy on test set: 0.839
SVM with RBF kernel - accuracy on test set: 0.944

C:\Users\irena\Anaconda3\lib\site-packages\sklearn\svm\base.py:193: FutureWarning: The default value of
gamma will change from 'auto' to 'scale' in version 0.22 to account better for unscaled features. Set g
amma explicitly to 'auto' or 'scale' to avoid this warning.
  "avoid this warning.", FutureWarning)
```

5. Tuning SVM parameters

SVM classifiers are very sensitive to the values of the parameters. To demonstrate this we will use a simpler dataset - the **moons** dataset. It contsins data from 2 classes described with 2 features; the data points form 2 half circles (moons). Let's generate and plot the data:

In [6]:
```
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=100, noise=0.15, random_state=42)

def plot_dataset(X, y, axes):
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs")
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "g^")
    plt.axis(axes)
    plt.grid(True, which='both')
    plt.xlabel(r"$x_1$", fontsize=20)
    plt.ylabel(r"$x_2$", fontsize=20, rotation=0)

plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])
plt.show()
```

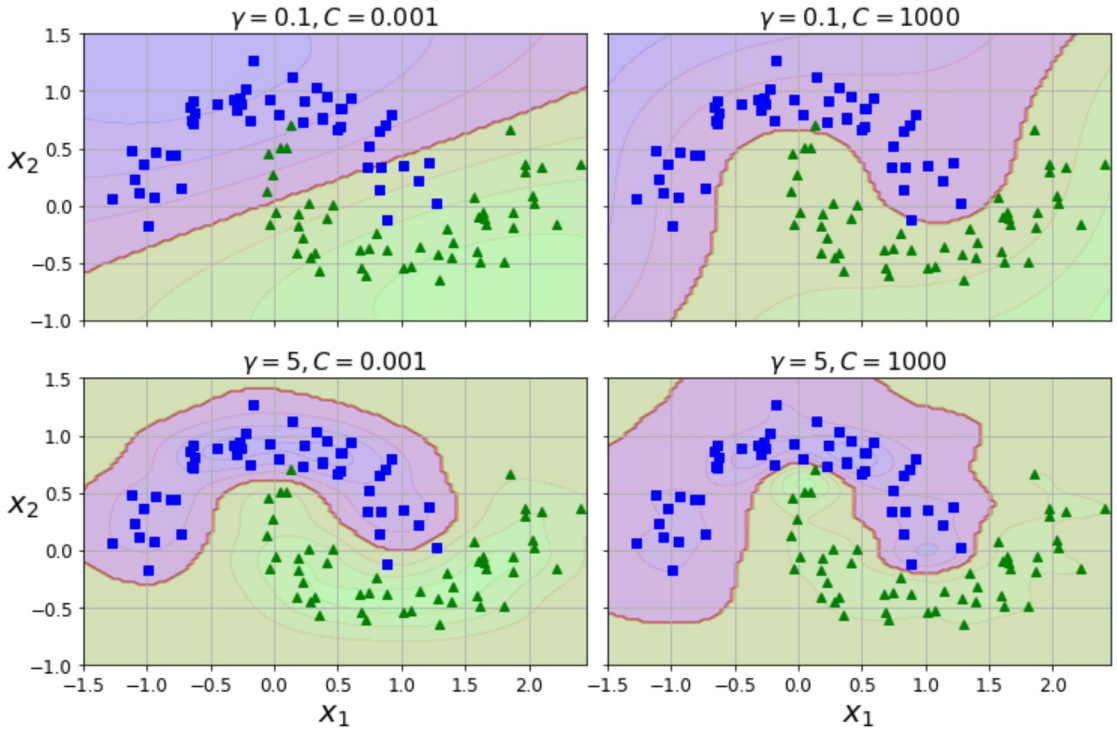


Now let's run SVM with RBF kernel on the moons dataset, with different values of the gamma and C parameters (the most important parameters) and observe how the decision boundary changes.

The parameter C controlds the regularization while the parameter gamma controls the width of the Gaussian kernel - smaller gamma means larger width and vice versa.

In [7]:
```
# Helper function for plotting the decision boundary
def plot_predictions(clf, axes):
    x0s = np.linspace(axes[0], axes[1], 100)
    x1s = np.linspace(axes[2], axes[3], 100)
    x0, x1 = np.meshgrid(x0s, x1s)
    X = np.c_[x0.ravel(), x1.ravel()]
    y_pred = clf.predict(X).reshape(x0.shape)
    y_decision = clf.decision_function(X).reshape(x0.shape)
    plt.contourf(x0, x1, y_pred, cmap=plt.cm.brg, alpha=0.2)
    plt.contourf(x0, x1, y_decision, cmap=plt.cm.brg, alpha=0.1)

# Create SVM with different gamma and C values
gamma1, gamma2 = 0.1, 5
C1, C2 = 0.001, 1000
hyperparams = (gamma1, C1), (gamma1, C2), (gamma2, C1), (gamma2, C2)

svm_clfs = []
for gamma, C in hyperparams:
    rbf_kernel_svm_clf = Pipeline([
            ("scaler", StandardScaler()),
            ("svm_clf", SVC(kernel="rbf", gamma=gamma, C=C))
        ])
    rbf_kernel_svm_clf.fit(X, y)
    svm_clfs.append(rbf_kernel_svm_clf)

fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10.5, 7), sharex=True, sharey=True)

# Plot the datset and decision boundary
for i, svm_clf in enumerate(svm_clfs):
    plt.sca(axes[i // 2, i % 2])
    plot_predictions(svm_clf, [-1.5, 2.45, -1, 1.5])
    plot_dataset(X, y, [-1.5, 2.45, -1, 1.5])
    gamma, C = hyperparams[i]
    plt.title(r"$\gamma = {}, C = {}$".format(gamma, C), fontsize=16)
    if i in (0, 1):
        plt.xlabel("")
    if i in (1, 3):
        plt.ylabel("")

save_fig("moons_rbf_svc_plot")
plt.show()
```

```
Saving figure moons_rbf_svc_plot
```



Going from top to bottom, the value of **gamma** increases:

- a small **gamma** (top figures) means a large radius of the bell-shaped RBF curve, i.e. the instances have a large receptive field and influence. As a result, the decision boundary is smoother, almost a line in the top-left figure.
- a big **gamma** (bottom figures) means a narrower bell-shape, i.e. each instance has a smaller receptive field and influence. As a result, the decision bountary is more irregular, wiggling around the individual instances.

Hence, **gamma** acts as a regularization parameter - if the SVM model is overfitting, **gamma** should be reduced; conversely, if it is underfitting - **gamma** should be increased.

Going from left to right, the value of **C** increases. Similarly to logistic regression and linear models:

- small **C** means a very restricted model, where each point has a small influence. We can see this in the top left figure - the decision boundary is almost a line and the misclassified points do not have any influence on it.
- big **C** means a less restrictive model and a bigger influence of all points. We can see how the decision boundary bends correctly to clasifify the previously misclassified points (compare the left top and left bottom figures).

Hence, a smaller **C** should be used if the model overfits and a larger if it underfits.

Tuning the SVM parameters is very important. It is necessary to experiment with different types of kernels and other parameters, e.g. using cross validation and grid search.

Regarding the type of kernel - the rule of thumb is to try first a linear kernel. In addition to SVC(kernel="linear"), there is another option: using the class LinearSVC. LinearSVC is much faster that SVC(kernel="linear"), especially if the training set is large (has many features and many examples). Next, SVM with RBF kernel should be tried as it typically works well, and then other types of kernels.

6. Tasks: SVM

Task 1: Load the iris dataset. Create two SVM classifiers: linear and RBF, and evaluate their accuracy using a single training/test split. Do this for a chosen set of parameter values, e.g. the default.

In [8]:
```
# Answer:

# Load the dataset
from sklearn.datasets import load_iris
iris = load_iris()

X_train_iris, X_test_iris, y_train_iris, y_test_iris = train_test_split(
    iris.data, iris.target, stratify=iris.target, random_state=42)

# Normalise data
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(X_train_iris)
X_train_iris_norm = scaler.transform(X_train_iris)
X_test_iris_norm = scaler.transform(X_test_iris)

# Linear SVM
lin_svm_iris = SVC(kernel="linear")
lin_svm_iris.fit(X_train_iris_norm, y_train_iris)
y_pred_iris = lin_svm_iris.predict(X_test_iris_norm)
print("Linear SVM on iris data - accuracy on test set: {:.3f}".format(accuracy_score(y_test_iris, y_pred_
iris)))

# RBF SVM
rbf_svm_iris = SVC(kernel="rbf", gamma="auto")
rbf_svm_iris.fit(X_train_iris_norm, y_train_iris)
y_pred_iris = rbf_svm_iris.predict(X_test_iris_norm)
print("SVM with RBF kernel - accuracy on test set: {:.3f}".format(accuracy_score(y_test_iris, y_pred_iri
s)))
```

```
Linear SVM on iris data - accuracy on test set: 0.921
SVM with RBF kernel - accuracy on test set: 0.895
```

Task 2: Consider the RBF SVM classifier. Use grid search with cross-validation to select a good combination of values for **C** and **gamma** from the following values: C = {0.001, 0.01, 0.1, 1, 10, 100} and gamma = {0.001, 0.01, 0.1, 1, 10, 100}. Show the accuracy on the test set.

In [9]:
```
#Answer

# Create the parameter grid
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
              'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
print("Parameter grid:\n{}".format(param_grid))

# The data is already split into training and test - we did this in Task 1

# Use GridSearcCV on the training set
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
grid_search = GridSearchCV(SVC(), param_grid, cv=10,
                           return_train_score=True)

grid_search.fit(X_train, y_train)

# Accuracy on test set of the model with selected best parameters:
print("Test set score: {:.2f}".format(grid_search.score(X_test, y_test)))

# You can also show these results:
print("Best parameters: {}".format(grid_search.best_params_))
print("Best cross-validation score: {:.2f}".format(grid_search.best_score_))
print("Best estimator:\n{}".format(grid_search.best_estimator_))
```

```
Parameter grid:
{'C': [0.001, 0.01, 0.1, 1, 10, 100], 'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
Test set score: 0.90
Best parameters: {'C': 1, 'gamma': 0.001}
Best cross-validation score: 0.93
Best estimator:
SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=0.001, kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

7. Dimensionality Reduction using Principal Component Analysis

Principal Component Analysis (PCA) is a very popular method for dimensionality reduction.

Let's apply PCA to the breast cancer data. It has 30 features. We can use PCA to find a lower-dimensional representation of this dataset that preserves the essential information. We will then apply a classifier, e.g. 1-nearest neighbor, on the original and reduced datasets and compare the results.

We have already loaded the breast cancer dataset and split it into training and test set - see Sec. 3 above. To apply PCA, we instantiate the PCA object, find the principle components by calling the **fit** method (applied to the training set only), and then project the training and test sets into the hyperpane defined by the first **n** principal components by calling the **transform** method.

We have set the number of components to 2, so the dimensionality reduction is from 30 to 2 dimensions (features). Recall that principal components define a new coordinate system in which the first axis corresponds to the direction of the highest variance in data, the second axis is ortogonal to the first one and corresponds to the second highest variance in data, and so on. We can check that the dimensionality was reduced to 2 by printing the shape of the original and transformed data.

In [10]:
```
from sklearn.decomposition import PCA
# keep the first two principal components of the data
# fit PCA model to breast cancer training data
pca = PCA(n_components=2).fit(X_train)

# transform training and test data onto the first two principal components
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)

print("Original shape of training data: {}".format(str(X_train.shape)))
print("Reduced shape of training data: {}".format(str(X_train_pca.shape)))
```

```
Original shape of training data: (426, 30)
Reduced shape of training data: (426, 2)
```

Let's apply 1-nearest neighbor to the original (30-dimensional) and the reduced (2-dimensional) data. We can see that the accuracy is high in both cases and almost the same: 0.94 vs 0.92. Hence, PCA was able to find a two-dimensional representation where the two classes separate well (accuracy >90%) when using 1-nearest neighbor.

Although we didn't see an improvement in accuracy on this dataset, PCA is very useful for highly dimensional data and often results in improved accuracy.

In [11]:
```
from sklearn.neighbors import KNeighborsClassifier

# Build a KNeighborsClassifier with using one neighbor:
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)
print("Original data - 1-nn accuracy: {:.2f}".format(knn.score(X_test, y_test)))

knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train_pca, y_train)
print("Reduced data - 1-nn accuracy: {:.2f}".format(knn.score(X_test_pca, y_test)))
```

```
Original data - 1-nn accuracy: 0.94
Reduced data - 1-nn accuracy: 0.92
```

8. Choosing the right number of dimensions in PCA

In the example above we chose to reduce the dimemsnionality to 2. Instead of arbitrary choosing the number of dimensions for the dimensionality reduction, we can specify the percentage of variance that we would like to preserve.

The variable **explained_variance_ratio_** shows the proportion of the dataset variance that lies along each ptincipal component. For our dataset: 98.2% of the dataset's variance lies along the first principal compoinent and 1.65% lies along the second principal component. This leaves very little for the third and other principal componnets (0.25%) so they will not be very informative. We can also see that most of the information is preserved by the first principle component.

In [12]:
```
pca.explained_variance_ratio_
```

Out[12]:
```
array([0.98197895, 0.01649533])
```

We can specify the percentage of variance we want to preserve, which will determine how many principal components to use to preserve this variance:

In [13]:
```
pca=PCA(n_components=0.95)
X_train_reduced = pca.fit_transform(X_train)

print("Reduced shape of training data: {}".format(str(X_train_reduced.shape)))
```

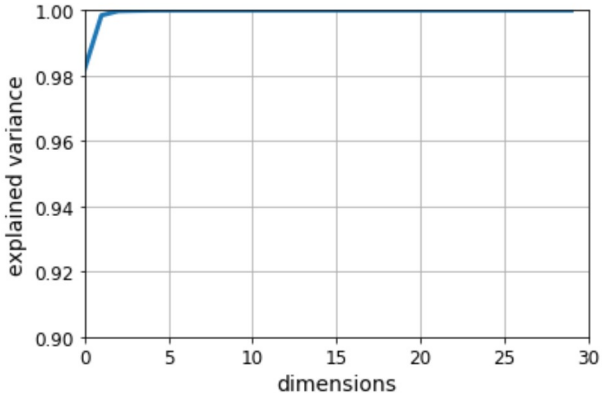
```
Reduced shape of training data: (426, 1)
```

Hence, 1 principal component is required to preseve 95% of the variance on the breast cancer datset.

Another option is to plot the explained variance vs the number of dimensions and inspect the graph visually. There will usually be an elbow in the curve, where the explained variance stops growing fast and this will determine the number of dimensions to use.

The following code shows how to plot the graph. The variance is plotted from 0.9 to 1 for a closer look at the elbow as for our dataset the first component captures >90% of the variance. The graph suggest that if we chose 1 or 2 dimensions we would lose very little explained variance.

In [14]:
```
# Perform PCA without reducing dimensionality
pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)

# Plot the explained variance vs number of dimensions

plt.figure(figsize = (6,4))
plt.plot(cumsum, linewidth=3)
#plt.axis([0, 30, 0, 1]) #[0,1] is the usual range for explained variance
plt.axis([0, 30, 0.9, 1]) #[0.9, 1] for closer look at the elbow
plt.xlabel("dimensions")
plt.ylabel("explained variance")
plt.grid(True)
plt.show()
```



9. Task: PCA for compression

PCA can also be applied for compression. We can demonstarte this on the MNIST digits dataset which contains 784 features.

Task 3:

1) Apply PCA to the MNIST dataset to reduce the number of features preserving 95% of the variance. How many features are needed? Plot the graph showing the explained variance vs the number of dimensions. Use the elbow method to select the number of features.

2) Decompress the reduced dataset back to 784 features using the **inverse_transform** method. Plot some digits from the the original dataset and their corresponding compressed version and observe the difference. Use the **plot_digits** function to plot the digits.

Solution:

To help you get started, we have loaded the MNIST dataset. The next step is to split it into training and test set and apply PCA.

In [15]:
```
# load the MNIST dataset
from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784', version=1)
mnist.target = mnist.target.astype(np.uint8)
```

In [16]:
```
# Create the training and test sets
from sklearn.model_selection import train_test_split

X_mnist = mnist["data"]
y_mnist = mnist["target"]

X_train_mnist, X_test_mnist, y_train_mnist, y_test_mnist = train_test_split(X_mnist, y_mnist)
```

Let's first apply PCA without reducing the dimensionality and compute the minimum number of dimensions (features) required for preserving 95% of the variance:

In [17]:
```
# apply PCA without reducing dimensionality, then compute the min number of dimensions for preserving 95% variance
pca = PCA()
pca.fit(X_train_mnist)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1
d
```

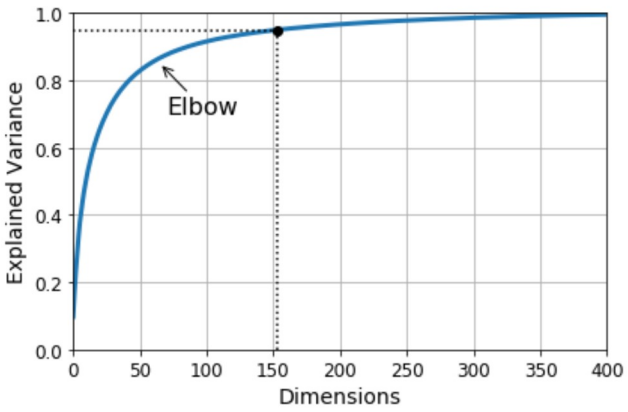
Out[17]: 153

We can see that 153 features are needed to preserve 95% of the variance. This is a considerable compression - from 784 to 153 features - we are using only 20% of the original features.

Now let's plot the explained variance graph and use the elbow method to find the reduced number of features. We can confirm that having around 150 features is a good choice since after that the explained variance doesn't grow fast (the graph flattens).

In [18]:
```
# Plot explained variance vs number of dimensions
plt.figure(figsize=(6,4))
plt.plot(cumsum, linewidth=3)
plt.axis([0, 400, 0, 1])
plt.xlabel("Dimensions")
plt.ylabel("Explained Variance")
plt.plot([d, d], [0, 0.95], "k:")
plt.plot([0, d], [0.95, 0.95], "k:")
plt.plot(d, 0.95, "ko")
plt.annotate("Elbow", xy=(65, 0.85), xytext=(70, 0.7),
             arrowprops=dict(arrowstyle="->"), fontsize=16)
plt.grid(True)
save_fig("explained_variance_plot")
plt.show()
```

```
Saving figure explained_variance_plot
```



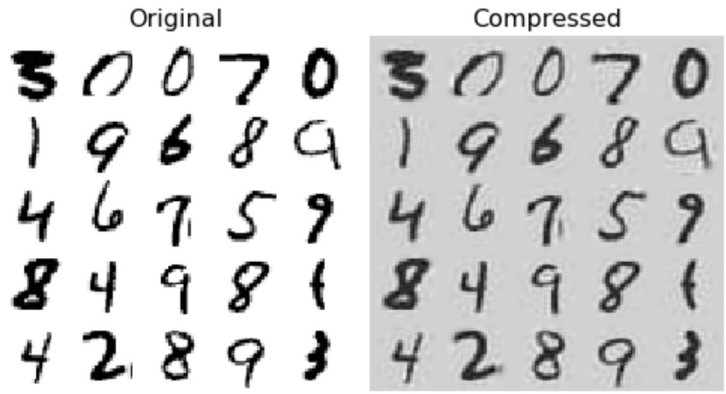
Now we can compress the MNIST dataset to 153 dimensions:

In [19]:
```
pca = PCA(n_components = 153)
X_reduced_mnist = pca.fit_transform(X_train_mnist)
X_recovered_mnist = pca.inverse_transform(X_reduced_mnist)
```

Finally, we can decompress the reduced dataset back to 784 dimensions by applying the inverse transformation of the PCA projection. This will not give us back the original data since we lost a bit of information (5% variance) but as we can see the image quality of the compressed dataset is pretty good:

In [20]:
```
plt.figure(figsize=(7, 4))
plt.subplot(121)
plot_digits(X_train_mnist[::2100])
plt.title("Original", fontsize=16)
plt.subplot(122)
plot_digits(X_recovered_mnist[::2100])
plt.title("Compressed", fontsize=16)

save_fig("mnist_compression_plot")
```

```
Saving figure mnist_compression_plot
```



Summary

In [21]:
```
lin_svm = SVC(kernel="linear")
lin_svm.fit(X_train_norm, y_train)
y_pred = lin_svm.predict(X_test_norm)
print("Linear SVM - accuracy on test set: {:.3f}".format(accuracy_score(y_test, y_pred)))

rbf_svm = SVC(kernel="rbf", gamma="auto")
rbf_svm.fit(X_train_norm, y_train)
y_pred = rbf_svm.predict(X_test_norm)
print("SVM with RBF kernel - accuracy on test set: {:.3f}".format(accuracy_score(y_test, y_pred)))

pca=PCA(n_components=0.95)
X_train_reduced = pca.fit_transform(X_train)
X_test_reduced = pca.transform(X_test)
print("Original shape of training data: {}".format(str(X_train.shape)))
print("Reduced shape of training data: {}".format(str(X_train_reduced.shape)))
```

```
Linear SVM - accuracy on test set: 0.979
SVM with RBF kernel - accuracy on test set: 0.944
Original shape of training data: (426, 30)
Reduced shape of training data: (426, 1)
```

Acknowledgements

This tutorial is based on:

Aurelien Geron (2019). Hands-on Machine Learning with Scikit-Learn, Keras and TensorFlow, O'Reilly.

Andreas C. Mueller and Sarah Guido (2016). Introduction to Machine Learning with Python: A Guide for Data Scientists, O'Reilly.