In [0]:
```
import matplotlib.pyplot as plt
import numpy as np
seed = np.random.seed
import pandas as pd
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

Preparación de datos

Utilizamos el conjunto de datos Iris del repositorio de aprendizaje automático UCI, que es uno de los conjuntos de datos más conocidos en clasificación. Consiste en 150 plantas de iris como ejemplos, cada una con los tamaños de sépalos y pétalos como atributos y el tipo como etiqueta de clase. Primero descarguemos el conjunto de datos de Iris usando Pandas:

In [0]:
```
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data', header = None)
df.columns = ['Sepal length', 'Sepal width', 'Petal length', 'Petal width', 'Class label']
display(df.head())

X = df[['Petal length', 'Petal width']].values
y = pd.factorize(df['Class label'])[0]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=0)

print('#Training data points: {}'.format(X_train.shape[0]))
print('#Testing data points: {}'.format(X_test.shape[0]))
print('Class labels: {} (mapped from {}'.format(np.unique(y), np.unique(df['Class label'])))
```

	Sepal length	Sepal width	Petal length	Petal width	Class label
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

```
#Training data points: 100
#Testing data points: 50
Class labels: [0 1 2] (mapped from ['Iris-setosa' 'Iris-versicolor' 'Iris-virginica']
```

Estandarización por el Gradient Descent Recall, el descenso del gradiente puede funcionar mal si el Hessian de la función f a minimizar tiene un número de condición grande.

En este caso, la superficie de f puede ser curvilínea en algunas direcciones pero plana en otras.

Por lo tanto, el descenso del gradiente, que ignora las curvaturas, puede sobrepasar el punto óptimo a lo largo de las direcciones curvas pero dar un paso demasiado pequeño a lo largo de las planas.

Una forma común de mejorar el condicionamiento de f es estandarizar su parámetro X, de la siguiente manera:

In [0]:
```
sc = StandardScaler()
sc.fit(X_train)
X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)
```

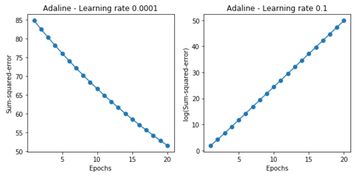
Implementando Adaline con GD

El ADAptive LInear NEuron (Adaline) es similar al Perceptron, excepto que define una función de costo basada en la salida de software y un problema de optimización. Por lo tanto, podemos aprovechar varias técnicas de optimización para entrenar a Adaline de una manera más teórica. Implementemos Adaline usando el algoritmo de descenso de gradiente por lotes (GD):

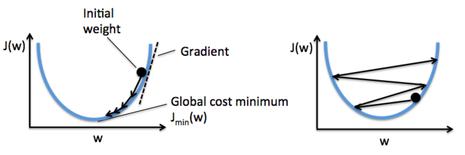
In [0]:
```
class AdalineGD(object):
    """ADAptive LInear NEuron classifier.
    Parámetros
    ------------
    eta: float        Tasa de aprendizaje (entre 0.0 y 1.0)
    n_iter: int       Pasa sobre el conjunto de datos de entrenamiento.
    random_state: int La semilla del generador de números pseudoaleatorios.
    Atributos
    -----------
    w_: 1d-array      Pesas después de la colocación.
    errores_: lista   Número de clasificaciones erróneas en cada época.
    """
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        """ Ajustar los datos de entrenamiento.
        Parámetros
        ----------
        X: {tipo matriz}, forma = [n_muestras, n_características]
            Vectores de entrenamiento, donde n_samples es el número de muestras y
            n_features es la cantidad de características.
        y: tipo matriz, forma = [n_muestras]
            Valores objetivo
        Devoluciones
        --------
        self: objeto
        """
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
        self.cost_ = []

        for i in range(self.n_iter):
            output = self.activation(X)
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()
            cost = (errors**2).sum() / 2.0
            self.cost_.append(cost)
        return self

    def net_input(self, X):
        """Calcular la net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, X):
        """Calcular activación lineal"""
        return self.net_input(X)

    def predict(self, X):
        """Etiqueta de clase de devolución después del paso de la unidad"""
        return np.where(self.activation(X) >= 0.0, 1, -1)
```

Como se discutió en la conferencia, una buena tasa de aprendizaje η es clave para la convergencia óptima. En la práctica, a menudo requiere algo de experimentación para encontrar una buena tasa de aprendizaje. Tracemos el costo contra el número de épocas para las dos tasas de aprendizaje diferentes:

In [0]:
```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(8, 4))

ada1 = AdalineGD(n_iter=20, eta=0.0001).fit(X_train_std, y_train)
ax[0].plot(range(1, len(ada1.cost_) + 1), ada1.cost_, marker='o')
ax[0].set_xlabel('Epochs')
ax[0].set_ylabel('Sum-squared-error')
ax[0].set_title('Adaline - Learning rate 0.0001')

ada2 = AdalineGD(n_iter=20, eta=0.1).fit(X_train_std, y_train)
ax[1].plot(range(1, len(ada2.cost_) + 1), np.log10(ada2.cost_), marker='o')
ax[1].set_xlabel('Epochs')
ax[1].set_ylabel('log(Sum-squared-error)')
ax[1].set_title('Adaline - Learning rate 0.1')

plt.tight_layout()
plt.show()
```

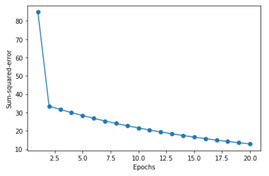


La figura de la izquierda muestra lo que podría suceder si elegimos una tasa de aprendizaje demasiado pequeña: aunque el costo disminuye, el descenso es demasiado pequeño para que el algoritmo requiera una gran cantidad de épocas para converger. Por otro lado, la figura correcta muestra lo que podría suceder si elegimos una tasa de aprendizaje que es demasiado grande: en lugar de minimizar la función de costo, el error se hace más grande en cada época porque sobrepasamos el punto óptimo cada vez. Esto se ilustra a continuación:



Con una tasa de aprendizaje correctamente elegida η, el AdalineGD converge y ofrece una mejor precisión de predicción (80%) en comparación con el Perceptron (70%):

In [0]:
```
ada = AdalineGD(n_iter=20, eta=0.01)
ada.fit(X_train_std, y_train)

# cost values

plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Sum-squared-error')

plt.tight_layout()
#plt.savefig('./output/fig-adalin-gd-cost.png', dpi=300)
plt.show()

# testing accuracy

y_pred = ada.predict(X_test_std)
print('Misclassified samples: %d' % (y_test != y_pred).sum())
print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
```




```
Misclassified samples: 31
Accuracy: 0.38
```

Como podemos ver, el valor del costo baja bastante rápido, y es apenas peor que el valor del costo (normalizado) del descenso del gradiente por lotes después de 7 épocas.

Otra ventaja del descenso de gradiente estocástico es que podemos usarlo para el aprendizaje en línea. En el aprendizaje en línea, un modelo se entrena sobre la marcha a medida que llegan nuevos datos de capacitación. Esto es especialmente útil si estamos acumulando grandes cantidades de datos con el tiempo. Por ejemplo, datos de clientes en aplicaciones web típicas. Mediante el aprendizaje en línea, el sistema puede adaptarse de inmediato a los cambios sin necesidad de capacitación desde cero. Además, si el espacio de almacenamiento es un problema, podemos descartar los datos de entrenamiento después de actualizar el modelo. En nuestra implementación, proporcionamos el método partial_fit() para el aprendizaje en línea.