project

December 18, 2015

1 Proyecto de Máquinas de Aprendizaje - ILI393

1.1 *Identificación facial cuando existen pocas muestras por clase*

1.2 El Problema

El problema consiste en la correcta identificación y reconocimiento facial, dado el caso particular en donde existen muy pocas muestras por clase (por persona) para entrenar los algoritmos algoritmos de clasificación. La mayoría de los enfoques usados en este tipo de problemas consisten en encontrar una buena representación de las características importantes de las caras, para posteriormente realizar algún tipo de búsqueda (jerárquica, nearest neighbors, etc). Sin embargo se propone aquí resolver el problema con tres algoritmos de clasificación distintos: Linear Discriminant Analysis, Support Vector Machines (con kernel lineal y radio basal) y por Convolutional Neural Networks.

Los dataset a ocupar son Faces94, Faces95 y Faces96. Cada uno de los datasets consiste en 20 imágenes de individuos, y variable cantidad de individuos. Los datasets están ordenados en cuanto a su complejidad de reconocimiento de menor a mayor. Se muestran a continuación imágenes representativas de Faces94, Faces95 y Faces96 respectivamente.

Para cada uno de estos datasets, se crearon los 20 training y testing sets correspondientes. La metodología fue la siguiente: Para cada dataset (94,95,96) se tomaron aleatoriamente entre 2-5 fotos por clase (por persona) para formar los training sets correspondientes, y las restantes 15-18 fotos se dejaron para crear los testing sets correspondientes. Vale decir, si se entrena con train94/5pc (faces94, 5 samples per class) entonces se prueba con test94/15pc (faces94, 15 samples per class). Cumpliendo de este modo con la restricción de tener pocos samples para entrenar los algoritmos. **Observación**: Por el momento se trabaja sólo con Faces94 y Faces95.

Para la comparación entre los resultados de los distintos algoritmos, se realizan *Error Bars* del error rate sobre los 20 datasets. Para computar las barras de error, se computan según el intervalo de confianza del 95% para la media del error, por medio de la distribución t-student.

1.3 Enfoque 1: Linear Discriminant Analysis

La implementación ocupada corresponde la de Scikit-Learn. Para cada una de las clases (personas) genera la función discriminante lineal δ_k, que permite diferenciar a cada una de las clases. Los dos supuestos fuertes que se realizan sobre los datasets al ocupar este método, son que 1) La probabilidad multivariada de las características $P(x_m|y = k)$ se distribuye normal, y que 2) La matriz de covarianza para cada una de las clases es igual.

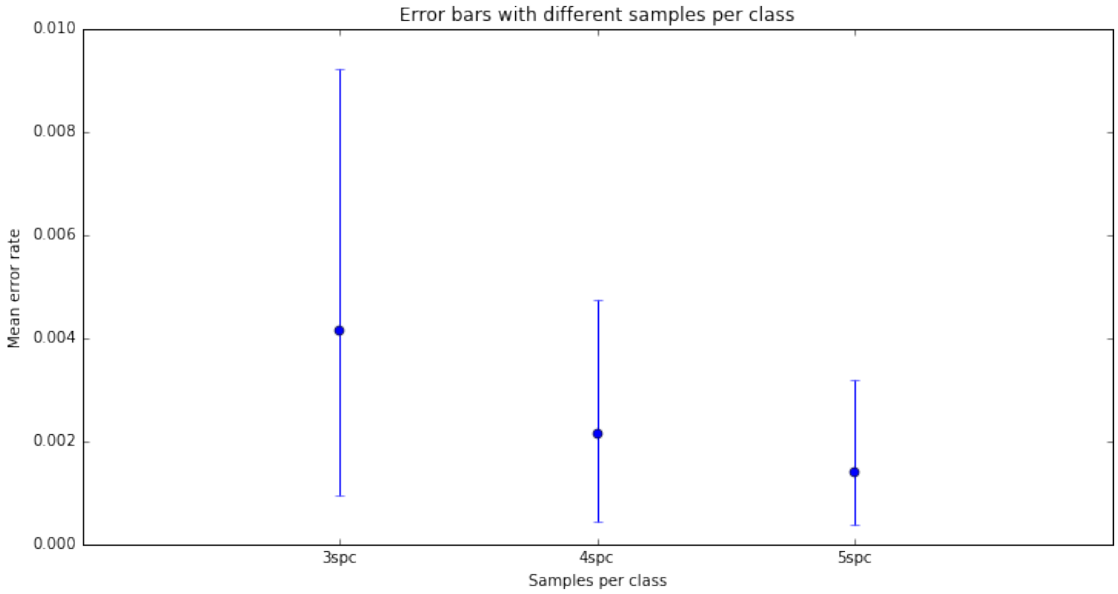
Debido a que LDA es un modelo generativo sin *hiperparámetros*, es que no es necesario realizar cross-validation para el modelo. Esto es una gran ventaja, pues ahora gran tiempo de computación (comparado con los otros métodos). Sin embargo, como se verá más adelante, paga este costo de simplicidad, entregando resultados menos precisos y generalizantes.

1.3.1 1) LDA con Faces94

```
In [85]: err_faces94_3spc = solve_lda('faces94', 3, verbose=False)
         err_faces94_4spc = solve_lda('faces94', 4, verbose=False)
```

```
err_faces94_5spc = solve_lda('faces94', 5, verbose=False)
err_faces94_6spc = solve_lda('faces94', 6, verbose=False)
errors_lda_faces94 = [err_faces94_3spc, err_faces94_4spc, err_faces94_5spc]
```

In [86]:
```
plot_errorbars(errors_lda_faces94)
```

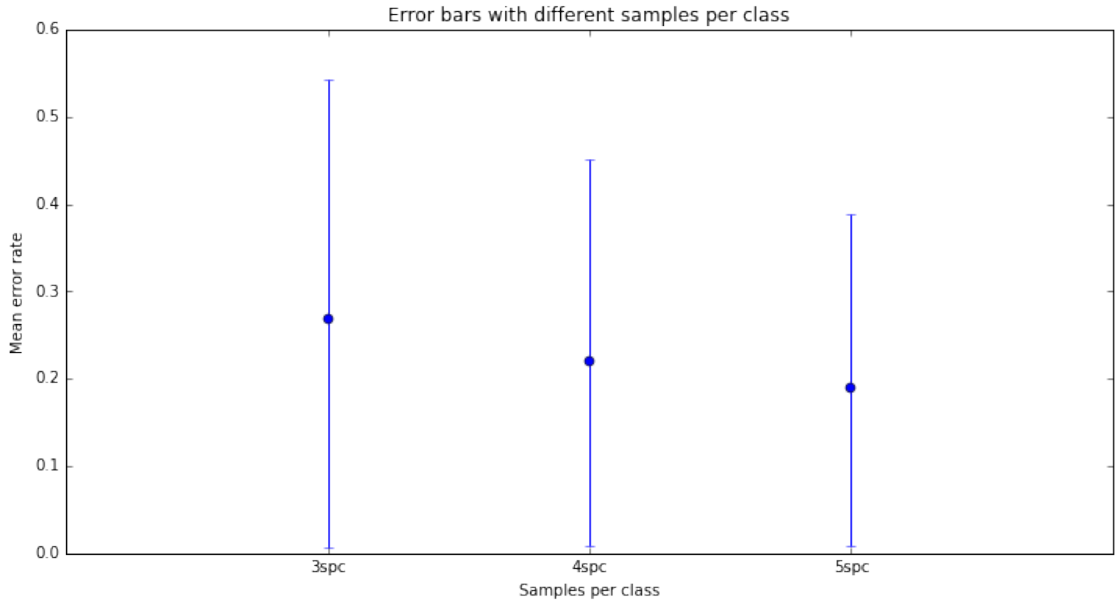


Análisis: + De los gráficos, se nota que el dataset Faces94 es muy "fácil" y por lo tanto los resultados son casi perfectos. Si se revisan las imagenes en este, puede notarse que las imagenes para una misma persona son todas muy parecidas, es decir, hay poca variación en la pose, expresión, iluminación, etc. Por lo cual esta data es fácilmente separable por hiperplanos. + De todos modos, a medida que aumentan las muestras por clase, el error rate tiende a disminuir.

1.3.2 LDA con Faces95

In [83]:
```
err_faces95_3spc = solve_lda('faces95', 3, verbose=False)
err_faces95_4spc = solve_lda('faces95', 4, verbose=False)
err_faces95_5spc = solve_lda('faces95', 5, verbose=False)
errors_lda_faces95 = [err_faces95_3spc, err_faces95_4spc, err_faces95_5spc]
```

In [84]:
```
plot_errorbars(errors_lda_faces95)
```



Análisis: + Se nota claramente que el dataset Faces95 es un dataset mucho más complejo que el anterior. + Los error rates son claramente mayores, siendo estos muy altos para los casos en donde hay pocos ejemplos por clase. + De todas maneras el comportamiento tiende a mejorar cuando existen más muestras por clase.

1.4 Enfoque 2: Support Vector Machines

1.4.1 Marco Teórico

Support Vector Machines (SVM) es un método de clasificación binario (adaptable a problemas de múltiples clases), que encuentra la frontera de decisión lineal óptima (hiperplano óptimo) que separa a las clases. Intuituvamente, una buena separación es lograda por el hiperplano con mayor distancia o margen a los datos de entramiento más cercanos. Esta superficie de separación es una combinación lineal de elementos del training set, conocidos como vectores de soporte, pues definen la frontera entre dos clases

Por lo tanto el método se reduce a el problema de optimización, de encontrar el hiperplano $\mathbf{w}^T x + b = 0$ que maximice el margen de cada ejemplo en el training set. Esto puede expresarse mas formalmente como:

$$\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2 \tag{1}$$

$$\text{s.t} \quad y_m(\mathbf{w}^T x_m + b) \geq 1, \ m = 1, \ldots, M \tag{2}$$

lo cual corresponde a un problema de optimización cuadrático. Es posible relajar algunas reestricciones del problema introduciendo *variables de holgura*

$$\min_{\mathbf{w},b,\zeta} \frac{1}{2}||\mathbf{w}||^2 + C\sum_{m=1}^{M} \zeta_i \tag{3}$$

$$\text{s.t} \quad y_m(\mathbf{w}^T \ x_m + b) \geq 1 - \zeta_m, \tag{4}$$

$$\zeta_m \geq 0, m = 1, ..., M \tag{5}$$

donde el parámetro C controla el trade-off entre el error de clasificación y el tamaño del margen. En general a mayor margen, menor será el error de generalización del clasificador, por lo tanto con las *variables de holgura* se tiene una manera de controlar esta relación.

La SVM determinada por el anterior problema de optimización es conocida como la C-SVM. Sin embargo por medio de una re-parametrización de esta, es posible definir un nuevo parámetros ν que controla el número de vectores de soporte y el error de entrenamiento. Esta formulación corresponde a la ν-SVM y puede ser demostrado que ambas formulaciones son matemáticamente equivalentes.

1.4.2 Experimentación

El enfoque seguido en esta sección, es la implementación y entrenamiento de multiclass SVM's con kernels tanto lineales como gaussianos, para cumplir con el objetivo de clasificar la data. Para ello se ocupan ν-SMV's, debido a que facilita la configuración del parámetro de holgura ν. La implementación ocupada corresponde a la de Scikit-Learn, el cual advierte: _"SVC implement one-against-one" approach (Knerr et al., 1990) for multi- class classification."_

Para la selección de *hiperparámetros* ν y γ (en kernels rbf) se realiza *stratified cross-validation* sobre cada training set, con la ayuda de grid search. Para que este método tenga sentido, el número de folds debe ser igual al número de muestras por clase (se hay más folds que muestras por clase, no se pueden cumplir las condiciones de estratificación).

Debido a que las dimensiones de las imágenes (200x180=36000) corresponden al total de features de cada foto, se ha decidido realizar una reducción de dimensionalidad para mejorar los tiempos de entrenamientos y eficiencia, así como también tomar las características realmente importantes (aquellas que permiten diferencias entre las clases).

Como técnica de reducción de dimensionalidad, se ha decidido ocupar LDA como reducción de dimensionalidad supervisada, representación tambien conocida como *Fisher Faces*. Esta técnica intenta encontrar un subespacio donde proyectar la data que permita diferenciar de mejor manera las clases. Dicho de otro modo, se intenta maximizar la **inter-class variance** y minimizar la varianza **intra-class variance**.

Básicamente los hiperplanos que conforman el subespacio de representación, corresponden a la funciones discriminantes que genera el modelo en LDA. La idea se plasma en la siguiente representación

en donde cada dato se proyecta en los hiperplanos discriminantes, para formar la representación. El número de máximo de discriminantes es $\min(\text{dimensiones}, \text{clases}-1)$, por lo tanto en un problema con muchas más dimensiones (features) que clases, la reducción de dimensionalidad es considerablemente favorable (este problema es precisamente el caso). Se pueden ocupar menos discriminantes, pero para los experimentos que se muestran a continuación se ocupan todos.

1.4.3 1) Linear SVM

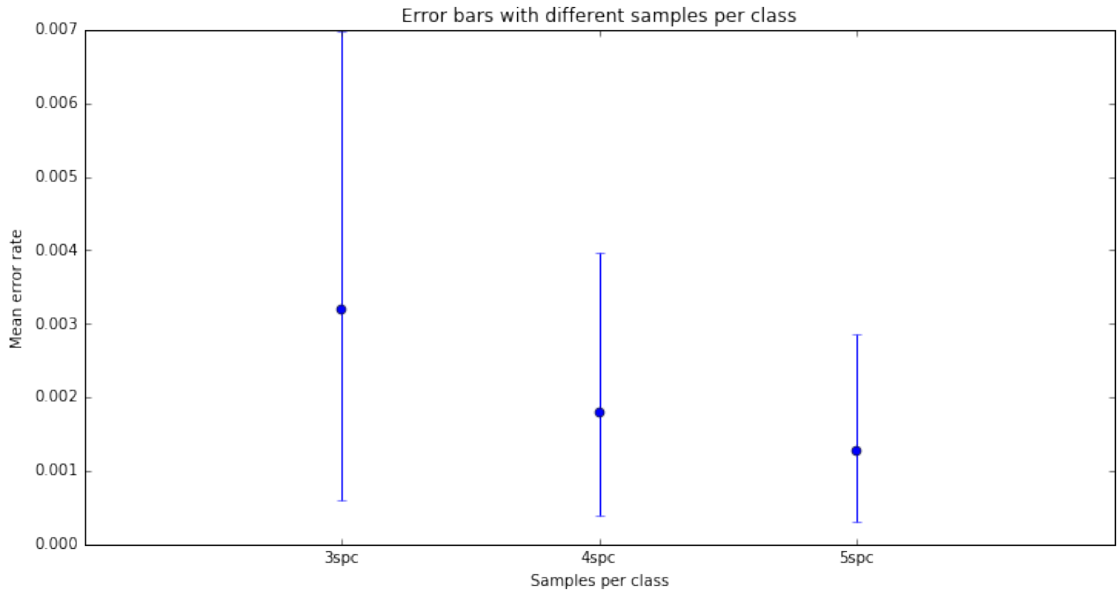
Observaciones: + Los rangos de los parámetros fueron determinados empíricamente, probando aquellos que entregan resultados coherentes y útiles en los datasets respectivos.

```
In [28]: #Setting parameters to try on Linear-SVM for Faces94 and Faces95
         Nu = np.linspace(0.0005, 0.001, 5, endpoint=True)
```

1.4.4 Linear-SVM con Faces94

```
In [81]: err_faces94_3spc = solve_svm('faces94', 3, 'linear', Nu, verbose=False)
         err_faces94_4spc = solve_svm('faces94', 4, 'linear', Nu, verbose=False)
         err_faces94_5spc = solve_svm('faces94', 5, 'linear', Nu, verbose=False)
         errors_lsvm_faces94 = [err_faces94_3spc, err_faces94_4spc, err_faces94_5spc]

In [119]: plot_errorbars(errors_lsvm_faces94)
```



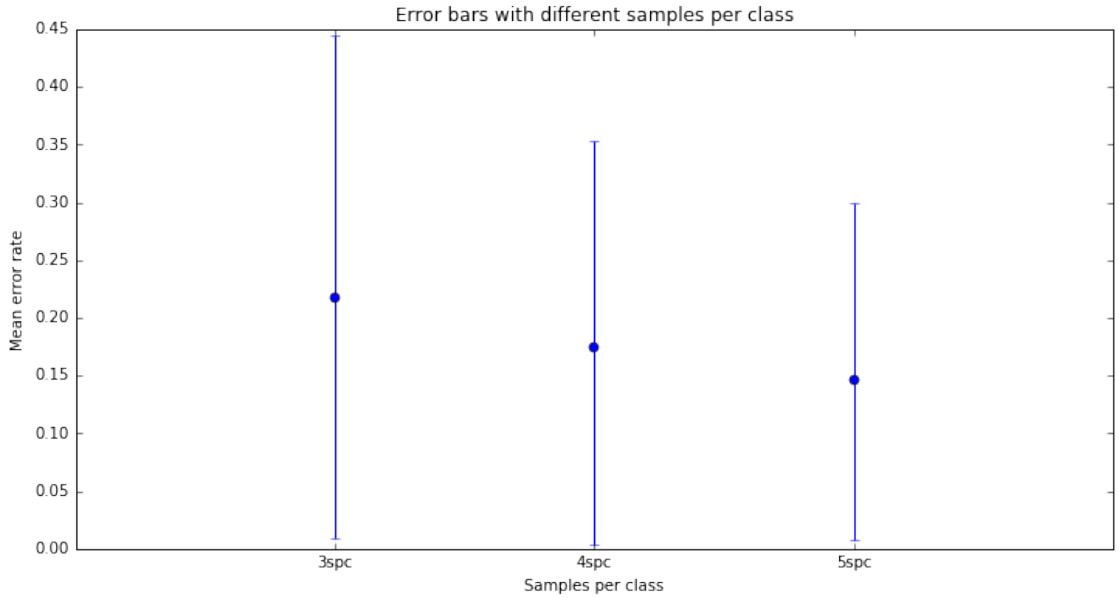
El gráfico a continuación compara los resultados recien obtenidos, con los resultados de los algoritmos anteriores para el dataset Faces94

Análisis: + Sucede lo mismo que con LDA: Faces94 es muy fácil de separar por hiperplanos, y por lo tanto obtiene un buen resultado también con la SVM de kernel lineal. + Los error rates obtenidos aquí son ligeramente inferiores a los obtenidos con LDA. + Sigue el patrón de mejorar los resultados a medida que aumentan las muestras por clase.

1.4.5 Linear-SVM con Faces95

```
In [76]: err_faces95_3spc = solve_svm('faces95', 3, 'linear', Nu, verbose=False)
         err_faces95_4spc = solve_svm('faces95', 4, 'linear', Nu, verbose=False)
         err_faces95_5spc = solve_svm('faces95', 5, 'linear', Nu, verbose=False)
         errors_lsvm_faces95 = [err_faces95_3spc, err_faces95_4spc, err_faces95_5spc]

In [80]: plot_errorbars(errors_lsvm_faces95)
```



El gráfico a continuación compara los resultados recien obtenidos, con los resultados de los algoritmos anteriores para el dataset Faces95

Análisis: + Los resultados aquí obtenidos, mejoran en una cantidad notoria respecto a el mismo dataset aplicado con LDA. Sin embargo, no hay que olvidar que aquí se tuvo que realizar un proceso de cross-validation para establecer *hiperparámetros* y en LDA no. + La mejor capacidad de generalización obtenida por la SVM, refleja los beneficios de tener variables de holgura, esto es, permitir cometer ciertos errores en el training set, para asi aumentar el margen junto con la capacidad de generalizar en el testing set. + Se mantiene el patrón de mejorar los resultados a medida que aumentan los ejemplos por clase.

1.4.6 2) Kernel-SVM

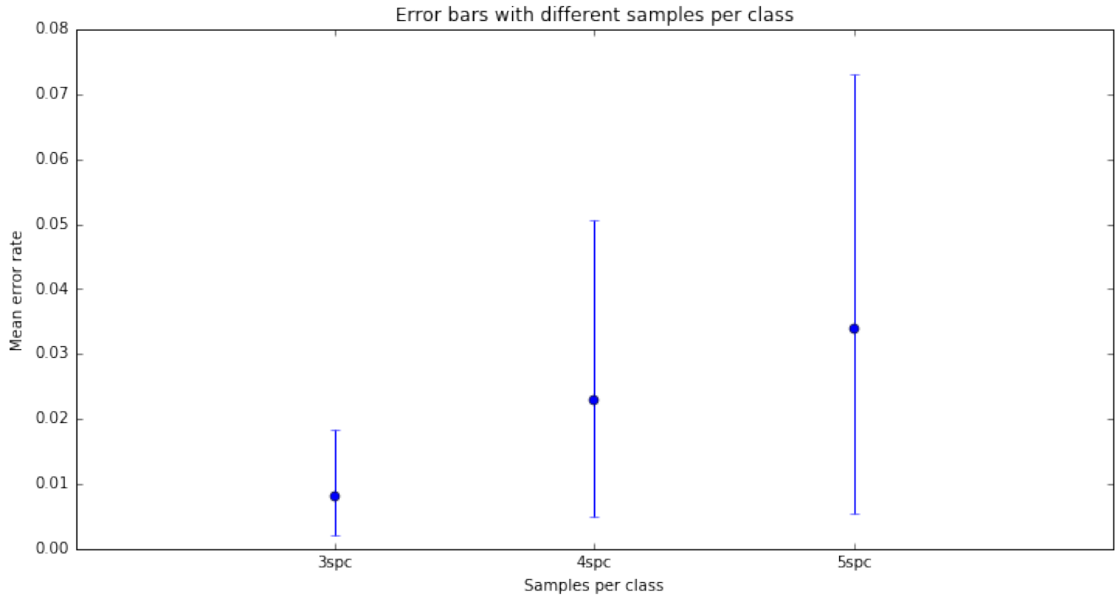
Observaciones: + Los rangos de los parámetros fueron determinados empíricamente, probando aquellos que entregan resultados coherentes y útiles en los datasets respectivos.

```
In [65]: #setting parameters to try one kernel-svm
         Nu = np.linspace(0.0005, 0.01, 10, endpoint=True)
         Gamma = np.linspace(0.25, 2.0, 10, endpoint=True)
```

1.4.7 Kernel-SVM con Faces94

```
In [ ]: err_faces94_3spc = solve_svm('faces94', 3, 'rbf', Nu, Gamma, verbose=False)
        err_faces94_4spc = solve_svm('faces94', 4, 'rbf', Nu, Gamma, verbose=False)
        err_faces94_5spc = solve_svm('faces94', 5, 'rbf', Nu, Gamma, verbose=False)
        errors_ksvm_faces94 = [err_faces94_3spc, err_faces94_4spc, err_faces94_5spc]
```

```
In [79]: plot_errorbars(errors_ksvm_faces94)
```



El gráfico a continuación compara los resultados recien obtenidos, con los resultados de los algoritmos anteriores para el dataset Faces94

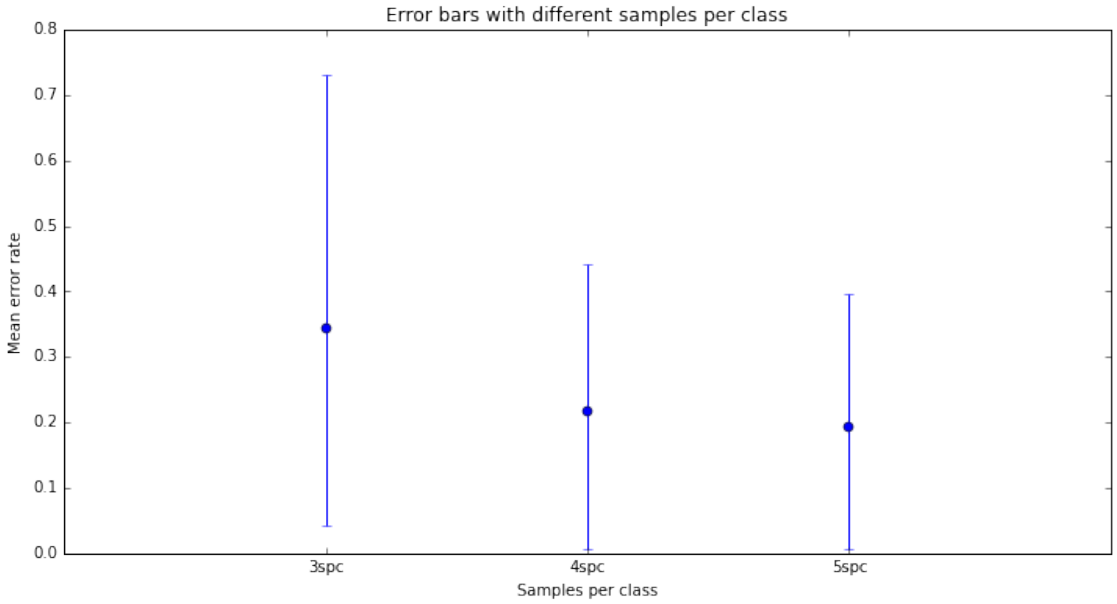
Análisis: + Sorprendentemente, los resultados son signigicativamente peores que con los otros dos algoritmos anteriores. Peor aun, el error en vez de disminiur a medida que aumentan las muestras por clase, aumenta. + La explicación más probable de este fenómeno, es overfitting. Como sabemos, con un kernel gaussiano se da la posibilidad de aprender fronteras de decisión no lineales complejas. Sin embargo sabemos que este dataset es muy bien separable por hiperplanos, por lo tanto el hecho de agregarle complejidad, sólo empeoró la situación. + Adicionalmente el hecho de que el error aumente con las muestras por clase, es un reflejo de lo descrito anteriormente. Mientras más muestras por clase existe, va a generar un modelo no lineal más y más complejo, lo cual para efectos de este dataset, es algo malo.

1.4.8 Kernel-SVM con Faces95

```
In [71]: #setting parameters to try one kernel-svm
         Nu = np.linspace(0.0005, 0.01, 10, endpoint=True)
         Gamma = np.linspace(0.25, 2.0, 10, endpoint=True)
```

```
In [ ]: err_faces95_3spc = solve_svm('faces95', 3, 'rbf', Nu, Gamma, verbose=False)
        err_faces95_4spc = solve_svm('faces95', 4, 'rbf', Nu, Gamma, verbose=False)
        err_faces95_5spc = solve_svm('faces95', 5, 'rbf', Nu, Gamma, verbose=False)
        errors_ksvm_faces95 = [err_faces95_3spc, err_faces95_4spc, err_faces95_5spc]
```

```
In [78]: plot_errorbars(errors_ksvm_faces95)
```



El gráfico a continuación compara los resultados recien obtenidos, con los resultados de los algoritmos anteriores para el dataset Faces95

Análisis: + Los resultados no son mejores que los obtenidos con los algoritmos anteriores, pero sin embargo no son tan malos como resulto en Faces94. Esto pues Faces95 es sabido un dataset más complejo, y por lo tanto un modelo de mayor complejidad (que uno lineal) puede ayudar. + Los resultados podrían mejorar aún más si se realiza las selección de *hiperparámetros* en una malla más fina, pero esto requeriría de mucho tiempo de computación.

1.5 Enfoque 3: Convolutional Neural Network

1.6 Enfoque 4: Dissimilarity SVM

1.6.1 Marco Teórico

Representación El problema de *identificación* tratado corresponde a dada una imagen de prueba $\mathbf{p}$, determinar a qué clase del conjunto de entrenamiento S_{tr} corresponde. Los enfoques basado en SVM anteriores, reducen este problema a K-class classification.

El propósito del método planteado, es ocupar las capacidades de clasificación de las SVM como clasificador binario, sobre dos conjuntos C_1 y C_2, donde el primero corresponde al *within-class differences set* que contienen las disimilitudes entre datos de la misma clase, y el segundo es *between-class difference set* y contienen las disimilitudes entre datos de distinta clase. El espacio en el que habitan los elementos de estos conjuntos, es conocido como el *difference space*, y contrasta con el espacio standard de las imágenes conocido como *image space*.

Formalizando lo anterior; sea $S_{tr} = \{\mathbf{s}_1, \ldots, \mathbf{s}_M\}$ el conjunto de entrenamiento con imágenes faciales de K individuos. Para indicar que dos individuos pertenecen a la misma clase ocuparemos $\mathbf{s}_i \sim \mathbf{s}_j$, y en caso contrario $\mathbf{s}_i \nsim \mathbf{s}_j$. Se define adicionalmente la función de similitud $\phi : R^N \times R^N \to R^S$ con $S \leq N$, como aquella función que mapea dos images, hacia el *difference space*. Luego es posible definir

$$C_1 = \{\phi(\mathbf{s}_i, \mathbf{s}_j) \mid \mathbf{s}_i \sim \mathbf{s}_j\} \tag{6}$$
$$C_2 = \{\phi(\mathbf{s}_i, \mathbf{s}_j) \mid \mathbf{s}_i \nsim \mathbf{s}_j\} \tag{7}$$

Entrenamiento Para el entrenamiento de la D-SVM (Dissimilarity SVM) los conjuntos de entrada son C_1 y C_2, es decir, se realiza un simple entrenamiento para clasificación binaria, lo cual es bastante conveniente.

Adicionalmente, sabemos que el output de una SVM es un conjunto de M_s vectores de soporte $\mathbf{v}_m$, coeficientes α_m, etiquetas de las clases y_m de los vectores de soporte y el término b constante. Luego la superficie de decisión puede ser escrita como

$$f(\mathbf{x}) = \sum_{m}^{M_s} \alpha_m y_m K(\mathbf{v}_m, \mathbf{x}) + b = 0$$

donde $K(\cdot, \cdot)$ es una función de kernel de acuerdo al Mercer's Theorem. Para $f(\mathbf{x}_i) < 0$, mientras más grande sea el valor de $|f(\mathbf{x_i})|$, más es el grado (o probabilidad) de pertenencia de $\mathbf{x}_i$ a la primera clase (de modo análogo se concluye para la segunda clase). Por lo tanto se puede ocupar la función f como discriminante, o como score de pertenencia a una clase.

1.7 Anexo de Código

1.7.1 Configuración del notebook

```
In [55]: %matplotlib inline
         import os
         import time
         import numpy as np
         import scipy as sp

         import matplotlib.pyplot as plt
         import cv2 as cv
         from sklearn.svm import SVC
         from sklearn.cross_validation import StratifiedKFold
         from sklearn.metrics import confusion_matrix
         from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
         from sklearn import svm, grid_search

         #libraries for convolutional neural network
         import theano.tensor as T
         import theano
         import lasagne
         from lasagne import layers
         from lasagne.updates import nesterov_momentum

         #libraries for proposed approach
         import histogram
         import metric
         from skimage.feature import local_binary_pattern as lbp
```

1.7.2 Helper functions

```
In [118]: """
          > function to load data from path directory to a matrix.
          > each row of the resulting matrix, corresponds to a flattened image
            in grayscale format
          """
          def load_data(path, spc, stacked=False):
              #total number of classes
              M = len(os.listdir(path))
              #dimensions of each image
```

```
    N = 200*180
    #matrix with features
    if stacked:
        data = np.empty((M*spc,1,200,180))
    else:
        data = np.empty((M*spc,N))
    labels = np.empty(M*spc)
    #index of data matrix
    m = 0
    for i in range(1,M+1):
        tgt = path+str(i)+'/'
        pics = os.listdir(tgt)
        for pic in pics:
            if stacked:
                #store each image, as a bidimensional array in data matrix
                data[m,0,:,:] = cv.imread(tgt+pic, cv.IMREAD_GRAYSCALE)
            else:
                #store each flattened image, as a row in data matrix
                data[m,:] = cv.imread(tgt+pic, cv.IMREAD_GRAYSCALE).ravel()
            labels[m] = i
            m += 1
    return (data.astype(np.float32), labels.astype(np.uint8))

def plot_confusion_matrix(cm, title='Confusion matrix', cmap=plt.cm.Blues):
    plt.figure(figsize=(15,8))
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.show()

def mean_confidence_interval(data, confidence=0.95):
    n = data.shape[0]
    #mean and standard error of mean
    mu, sem = np.mean(data), sp.stats.sem(data)
    #computing confidence interval
    h = sem * sp.stats.t.ppf((1+confidence)/2., n-1)
    return mu, mu-h, mu+h

def plot_errorbars(errors):
    mean = []
    lower = []
    upper = []
    for error in errors:
        mu,l,u = mean_confidence_interval(error)
        mean.append(mu)
        lower.append(l)
        upper.append(u)
    x = np.arange(3, len(errors)+3)
    mean = np.array(mean)
    lower = np.array(lower)
    upper = np.array(upper)
```

```
    labels = ['3spc','4spc','5spc']
    plt.figure(figsize=(12,6))
    plt.xlim([2,6])
    plt.xticks(x, labels)
    plt.errorbar(x, mean, yerr=[lower,upper],fmt='o')
    plt.xlabel('Samples per class')
    plt.ylabel('Mean error rate')
    plt.title('Error bars with different samples per class')
    plt.show()


def plot_comparative_errorbars(errors):
    pass

"""
This is just a simple helper function iterating over training data in
mini-batches of a particular size, optionally in random order. It assumes
data is available as numpy arrays. For big datasets, you could load numpy
arrays as memory-mapped files (np.load(..., mmap_mode='r')), or write your
own custom data iteration function. For small datasets, you can also copy
them to GPU at once for slightly improved performance. This would involve
several changes in the main program, though, and is not demonstrated here.
"""

def iterate_minibatches(inputs, targets, batchsize, shuffle=False):
    assert inputs.shape[0] == targets.shape[0]
    M = inputs.shape[0]
    num_batches = M/batchsize
    if shuffle:
        indices = np.arange(M)
        np.random.shuffle(indices)
    for batch_idx in range(num_batches):
        start_idx = batch_idx*batchsize
        if batch_idx==num_batches-1: end_idx = M
        else: end_idx = start_idx+batchsize
        if shuffle:
            excerpt = indices[start_idx:end_idx]
        else:
            excerpt = slice(start_idx, end_idx)
        yield inputs[excerpt], targets[excerpt]
```

1.7.3 Funciones de error

```
In [3]: def precision(y_ts, y_pd):
            #true positives
            tp = (y_ts==y_pd).sum()
            #total of predictions
            n = y_ts.shape[0]
            return tp/np.float(n)

        def error_rate(y_ts, y_pd):
            return 1-precision(y_ts, y_pd)
```

1.7.4 Funciones para LDA

```
In [4]: def solve_lda(dataset, spc, verbose=False):
            #samples per class on training set
            spc_tr = spc
            spc_ts = 20-spc_tr
            #training and testing paths
            tr_path = './db/train'+dataset[-2:]+'/tr-{0}pc-{1}/'
            ts_path = './db/test'+dataset[-2:]+'/ts-{0}pc-{1}/'
            #errors through all datasets
            err = list()
            #iterating through datasets
            for set_num in range(20):
                #loading training and testing set
                X_tr,y_tr = load_data(tr_path.format(spc_tr,set_num), spc_tr)
                X_ts,y_ts = load_data(ts_path.format(spc_ts,set_num), spc_ts)
                #creating LDA object and fitting the testing data
                clf = LDA()
                clf.fit(X_tr, y_tr)
                #making predictions
                y_pd = clf.predict(X_ts)
                #computing error rate
                err.append(error_rate(y_ts,y_pd))
                if verbose:
                    print "##############################################################################
                    print "{0}: {1} samples per class (dataset {2})".format(dataset, spc, set_num)
                    print "Error rate: {0}".format(err[-1])
                #releasing memory of big objects
                del X_tr, X_ts, clf
            return np.array(err)
```

1.7.5 Funciones para SVM

```
In [62]: """
         Supervised dimensionality reduction through LDA
         """
         def fisher_faces(X, y):
             #supervised learning through LDA
             #finding the discriminant functions
             ff = LDA()
             ff.fit(X,y)
             #proyect the data into linear discriminant hyperplanes
             return ff

         """
         Stratified 5-fold cross validation y Grid search
         para determinar el mejor parametro C en linear svm
         """
         def cross_linear_svm(X, y, Nu, n_folds=None):
             #generating stratified 5-fold cross validation iterator
             strat_kf = StratifiedKFold(y, n_folds=n_folds, shuffle=True)
             #parameters to try
             params = {'nu':Nu}
             #Setting grid search for linear-svm
             clf = svm.NuSVC(kernel='linear')
```

```
    gs = grid_search.GridSearchCV(clf, params, cv=strat_kf, n_jobs=2)
    #make it
    gs.fit(X, y)
    #return best parameters and grid scores
    return gs.best_params_['nu'] , gs.grid_scores_


"""
Stratified 5-fold cross validation y Grid search
para determinar el mejor parametro Nu y Gamma en rbf svm
"""
def cross_rbf_svm(X, y, Nu, Gamma, n_folds=None):
    #generating stratified 5-fold cross validation iterator
    strat_kf = StratifiedKFold(y, n_folds=n_folds, shuffle=True)
    #parameters to try
    params = {'nu':Nu, 'gamma':Gamma}
    #Setting grid search for rbf-svm
    clf = svm.NuSVC(kernel='rbf')
    gs = grid_search.GridSearchCV(clf, params, cv=strat_kf, n_jobs=2)
    #make it
    gs.fit(X, y)
    #return best parameters and grid scores
    return gs.best_params_['nu'], gs.best_params_['gamma'] , gs.grid_scores_


def solve_svm(dataset, spc, kernel, Nu, Gamma=None, verbose=False):
    #samples per class on training set
    spc_tr = spc
    spc_ts = 20-spc_tr
    #training and testing paths
    tr_path = './db/train'+dataset[-2:]+'/tr-{0}pc-{1}/'
    ts_path = './db/test'+dataset[-2:]+'/ts-{0}pc-{1}/'
    #errors through all datasets
    err = list()
    #iterating through datasets
    for set_num in range(20):
        #loading training and testing sets
        X_tr,y_tr = load_data(tr_path.format(spc_tr,set_num), spc_tr)
        X_ts,y_ts = load_data(ts_path.format(spc_ts,set_num), spc_ts)
        #projecting into discriminant space
        ff = fisher_faces(X_tr, y_tr)
        X_tr = ff.transform(X_tr)
        X_ts = ff.transform(X_ts)
        #choosing best nu (and gamma) through stratified
        #5-fold cross-validation and grid search
        if kernel=='linear':
            nu,grid_scores = cross_linear_svm(X_tr, y_tr, Nu, n_folds=spc)
        elif kernel=='rbf':
            nu,gamma,grid_scores = cross_rbf_svm(X_tr, y_tr, Nu, Gamma, n_folds=spc)
        #fitting the model
        if kernel=='linear':
            clf = svm.NuSVC(kernel='linear', nu=nu)
        elif kernel=='rbf':
            clf = svm.NuSVC(kernel='rbf', nu=nu, gamma=gamma)
```

```
            clf.fit(X_tr,y_tr)
            #making predictions
            y_pd = clf.predict(X_ts)
            #computing error rate
            err.append(error_rate(y_ts,y_pd))
            if verbose:
                print "##############################################################################
                print "{0}: {1} samples per class (dataset {2})".format(dataset, spc, set_num)
                print "Error rate: {0}".format(err[-1])
                print "Best Nu: {0}".format(nu)
                if kernel=='rbf':
                    print "Best gamma: {0}".format(gamma)
            #releasing memory of big objects
            del X_tr, X_ts, clf
        return np.array(err)
```

1.7.6 Funciones para Dissimilarity SVM

In [58]:
```
"""
Transform data on X matrix (flattened) to
spatial histograms representation on Y matrix
"""
def to_spatial_histogram(X, P=8, R=2, Nx=10, Ny=10, Npatterns=59, OverlapX=2, OverlapY=2):
    M,N = X.shape
    #return matrix
    Y = np.empty((M,59*Nx*Ny))
    for m in range(M):
        img = X[m].reshape((-1,180))
        lbp_img = lbp(img, P, R, method='nri_uniform')
        sp_hist = histogram.spatial(lbp_img, Nx, Ny, Npatterns, OverlapX, OverlapY)
        Y[m] = sp_hist
    return sp_hist
```