Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ingeniería Eléctrica

Tarea 2

Support Vector Machine

Integrantes: Vincko Fabres
Profesor: Javier Ruiz del Solar
Auxiliar: Patricio Loncomilla Z.
Ayudantes: Juan Pablo Cáceres B.
Pablo Troncoso P.
Rodrigo Salas O.
Rudy García
Sebastian Solanich

Fecha de entrega: 29 de abril de 2021
Santiago, Chile

Índice de Contenidos

Índice de Figuras

Índice de Tablas

Índice de Códigos

1. Introducción

En el campo de machine learning son variadas las técnicas de aprendizaje, esta experiencia busca un acercamiento a los support vector machine (SVM), dentro de los cuales existen diferentes tipos. Para realizar esta experiencia se utilizará la plataforma colab [1], programando con el lenguaje de programación Python y la utilización de las bibliotecas Pandas, sklearn, Matplotlib y Time. Para entrenar el clasificador y testeo de este se utilizará el conjunto de datos MAGIC Gamma Telescope Data Set, el cual corresponde a un conjunto de simulaciones de air showers generados por rayos gamma primarios v/s hadrones, este conjunto de datos posee 10 características, sin contar su clase; la cual puede ser hadrón o g no-hadrón.
La experiencia cuenta con 5 sub-bloques los cuales son:

- Visualización y análisis de datos
- Clasificación usando SVM lineal y rendimiento
- Clasificación usando SVM de kernel polinomial y rendimiento
- Clasificación usando SVM de kernel RBF y rendimiento
- Comparación de resultados

Las cuales son explicadas en en Desarrollo del informe para posteriormente ser analizadas y dar paso a la sección de Conclusiones.

[1] https://colab.research.google.com/

2. Desarrollo

2.1. Parte 1: Visualización y análisis de datos

La primera parte consta de la manipulación del conjunto de datos **magic04.data**, utilizando la función *pd.read_csv()* con esto se permite subir el archivo al entorno de ejecución.

Código 1: Subida de datos a Collab

```
#subir archivo magic04.data
from google.colab import files
uploaded= files.upload()
```

Debido a que posteriormente se utilizarán las bibliotecas Pandas, Matplotlib, Sklearn y Time, se deben importar, como sigue a continuación:

Código 2: Bibliotecas importadas

```
%matplotlib inline
import pandas as pd
import matplotlib.pyplot as plt
import sklearn
import time
```

2.1.1.

Seguida la inicialización del código se debe leer y manipular el conjunto de datos, nombrar a sus columnas con su respectiva característica, como sigue a continuación:

Código 3: Manipulación de dataset

```
nombres_columnas=['fLength', 'fWidth', 'fSize', 'fConc', 'fConc1', 'fAsym', 'fM3Long', 'fM3Trans', '
    ↪ fAlpha', 'fDist', 'class']
data = pd.read_csv("magic04.data",names=nombres_columnas)
df= pd.DataFrame(data)
df['class'].replace('g',0, inplace = True)
df['class'].replace('h',1, inplace = True)
reordenar = df
reordenar = reordenar.sample(frac=1,random_state=1).reset_index(drop=True)
clase_positiva=reordenar.loc[reordenar['class'] == 1] #hadrones
clase_negativa=reordenar.loc[reordenar['class'] == 0] #no hadrones
remuestreo= pd.concat([clase_positiva[:3000],clase_negativa[:3000]])
remuestreo = remuestreo.sample(frac=1,random_state=1).reset_index(drop=True)
```

Dónde la función del código realizado es remuestrear el dataset, para contar con 3000 ejemplares por clase de forma aleatoria.

Una vez obtenido el nuevo dataset se deben separar los conjuntos en 3, siendo estos: entrenamiento, validación y prueba.

Código 4: Creación de subconjuntos

```
num_entr = int(len(remuestreo)*0.6)
num_validacion = int(len(remuestreo)*0.2)
num_test= len(remuestreo)-num_entr-num_validacion

entrenamiento= remuestreo.iloc[0:num_entr]
validacion= remuestreo.iloc[num_entr:(num_entr+num_validacion)]
validacion= validacion.reset_index(drop=True)
test= remuestreo.iloc[(num_entr+num_validacion):]
test= test.reset_index(drop=True)

#proporcion conjunto completo
ratio_df= len(remuestreo.loc[remuestreo['class'] == 0])/len(remuestreo.loc[remuestreo['class'] == 1])
    ↪  #proporcion de clases #0/#1
#poporcion conj entrenamiento
ratio_entrenamiento=len(entrenamiento.loc[entrenamiento['class']==0])/len(entrenamiento.loc[
    ↪ entrenamiento['class']==1])
print((abs(ratio_df-ratio_entrenamiento)/ratio_df)*100 ,'% de diferencia en representatividad para
    ↪ entrenamiento') #formula de error
#poporcion conj validacion
ratio_validacion=len(validacion.loc[validacion['class']==0])/len(validacion.loc[validacion['class']==1])
print((abs(ratio_df-ratio_validacion)/ratio_df)*100 ,'% de diferencia en representatividad para
    ↪ validacion') #formula de error

#poporcion conj testeo
ratio_test=len(test.loc[test['class']==0])/len(test.loc[test['class']==1])
print((abs(ratio_df-ratio_test)/ratio_df)*100 ,'% de diferencia en representatividad para test') #
    ↪ formula de error
```

Este divide el dataset en 3, con el primer subconjunto se realizan los entrenamientos de los SVM, con el segundo se calibran los hiperparámetros y el tercero sirve de prueba, aunque en la realidad este último es una 'caja negra'.En cada subconjunto se debe analizar la representatividad de este mismo con respecto al dataset, para lo cual se analiza la proporción entre clases del dataset y posteriormente la proporción de clases para cada subconjunto, un valor representativo para ver qué tan fiable es la representación es utilizar la fórmula de error porcentual, el cual es el siguiente para cada subconjunto:

Tabla 1: Error porcentual de representatividad

Conjunto	error porcentual
Entrenamiento	0.33 %
Validación	4.24 %
Representatividad	3.39 %

Finalmente se debe entrenar un StandardScaler con el conjunto de entrenamiento, lo cual es ejecutado como sigue:

Código 5: Estandarización de características

```
from sklearn import preprocessing
scaler= preprocessing.StandardScaler()
scaler_entrenado=scaler.fit(entrenamiento.iloc[:,:9]) #Entrenamiento de standardscaler

#Aplicación de standardscaler a conjuntos
scaled_entrenamiento= scaler.transform(entrenamiento.iloc[:,:9])
scaled_validacion= scaler.transform(validacion.iloc[:,:9])
scaled_test= scaler.transform(test.iloc[:,:9])
```

Este estandariza los datos normalizando las características, dejandolas cercanas a varianza 0 y mediana 1, haciendo que las características queden similares para poder trabajarlas mejor.

3. Parte 2: Clasificación usando SVM lineal y rendimiento

Para utilizar SVM lineal se deben escoger los mejores hiper parámetros, razón por la cual se utiliza una grilla como se ve a continuación:

Código 6: Grilla SVM lineal

```
from sklearn import svm
from sklearn.model_selection import GridSearchCV
tiempo_svmlineal_inicio = time.time()


parametros= {'C':[0.1, 0.25, 0.5, 0.75, 1]}
svmlineal = svm.SVC(kernel= 'linear', probability=False)
clfl = GridSearchCV(svmlineal, parametros, cv=5)
clfl.fit(scaled_entrenamiento,entrenamiento.iloc[:,10])

tiempo_svmlineal_final = time.time()
print(clfl.best_params_)
```

Donde *clfl* tiene guardado al mejor SVM lineal según su grilla e hiper parámetros, igualmente la función *best_params_* retorna los mejores parámetros de la grilla.

Posteriormente para la evaluación de hiper parámetros se utiliza como test el conjunto de validación, del cual se obtiene la matriz de confusión. Esto se realiza mediante el código:

Código 7: Evaluación mediante matriz de confusión

```

y_pred_svml=clfl.predict(scaled_validacion)
print(sklearn.metrics.confusion_matrix(validacion.iloc[:,10], y_pred_svml, normalize=None))
print(sklearn.metrics.confusion_matrix(validacion.iloc[:,10], y_pred_svml, normalize='true'))
```

La cual retorna las siguientes matrices de confusión:

Tabla 2: Matriz de confusión

	Negativos Predichos	Positivos Predichos
Negativos Observados	472	115
Positivos Observados	161	452

La cual normalizada retorna los siguientes parámetros:

Tabla 3: Matriz de confusión normalizada

	Negativos Predichos	Positivos Predichos
Negativos Observados	0.80	0.20
Positivos Observados	0.26	0.74

Posteriormente se genera una curva ROC para mostrar el desempeño del clasificador, esto mediante el código:

Código 8: Curva ROC SVM lineal

```
1
2  from sklearn import metrics
3  y_score_svml=clf.decision_function(scaled_validacion)
4  #sklearn.metrics.roc_curve(y_true, y_score, *, pos_label=None, sample_weight=None,
       ↪ drop_intermediate=True)
5  fprl, tprl, thresholds = sklearn.metrics.roc_curve(validacion.iloc[:,10], y_score_svml)
6  area_bajo_curva_svml=metrics.auc(fprl, tprl)
7
8  plt.figure()
9
10 plt.plot(fprl, tprl, color='red', label='Curva ROC SVML (area = %0.2f)' % area_bajo_curva_svml)
11 plt.plot([0, 1], [0, 1], color='blue',  linestyle='--')
12 plt.xlim([0.0, 1.0])
13 plt.ylim([0.0, 1.05])
14 plt.xlabel('False Positive Rate')
15 plt.ylabel('True Positive Rate')
16 plt.title('Curva ROC SVM Lineal para validación')
17 plt.legend(loc="lower right")
18 plt.show()
```

El cual genera la siguiente curva ROC:

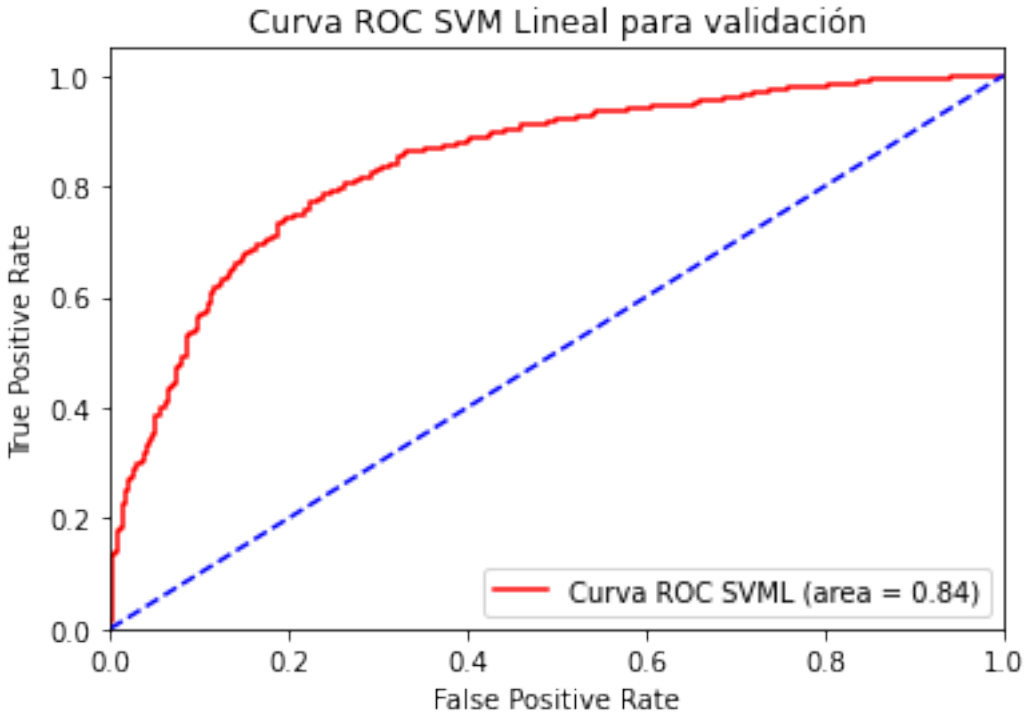

Figura 1: Curva Roc SVM lineal

Del cual podemos ver que el área bajo la curva es de 0.84

Por último se debe obtener su respectiva curva precision recall, siendo el código para implementarla el siguiente:

Código 9: Precision recall SVM lineal

```
1
2  precisionl, recalll, thresholds = sklearn.metrics.precision_recall_curve(validacion.iloc[:,10],
       ↪ y_score_svml)
3  avg_ps_svml=sklearn.metrics.average_precision_score(validacion.iloc[:,10], y_score_svml)
4
5  plt.figure()
6
7  plt.plot(recalll,precisionl, color='red', label='Precision Recall SVML (average precision = %0.2f)' %
       ↪ avg_ps_svml)
8  plt.plot([0, 1], [1,0], color='blue',  linestyle='--')
9  plt.xlim([0.0, 1.0])
10 plt.ylim([0.0, 1.05])
11 plt.xlabel('Recall')
12 plt.ylabel('Precision')
13 plt.title('Curva Precision Recall SVM Lineal para validación')
14 plt.legend(loc="lower left")
15 plt.show()
```

El que retorna la siguiente gráfica:

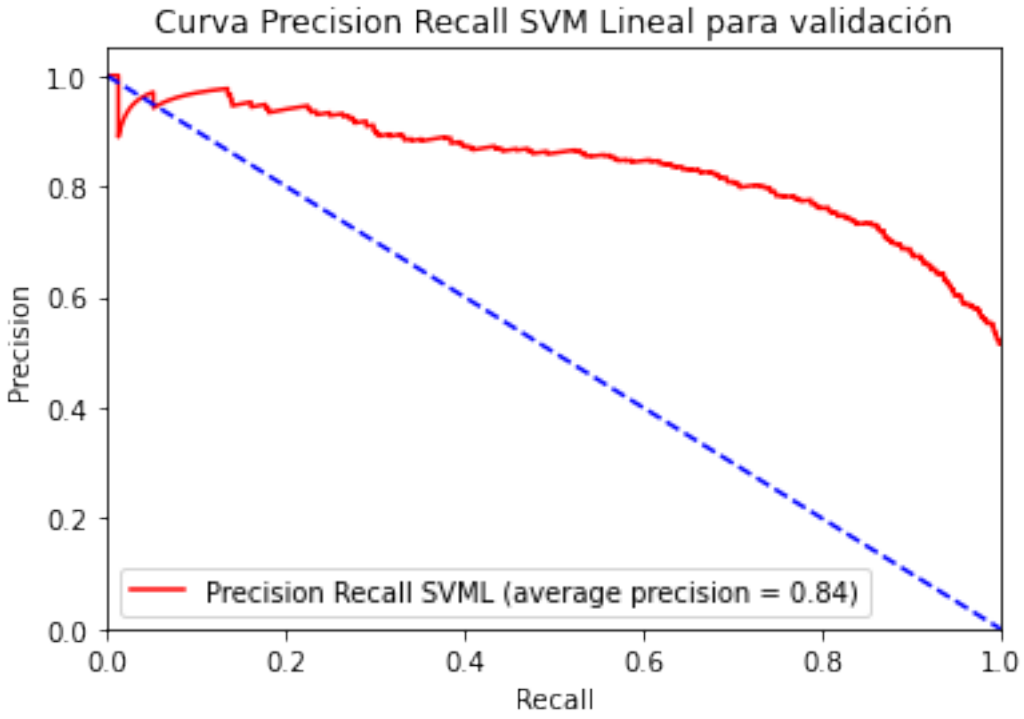

Figura 2: Curva precision recall SVM lineal

Del cual podemos notar que su average precision es de 0.84.

4. Parte 3: Clasificación usando SVM de kernel polinomial y rendimiento

Para utilizar un SVM de kernel polinomial lo primero es escoger hiper parámetros, razón por la cual se implementa una grilla, la cual consta de 4 valores para 'C', 4 para 'Gamma' y se implementa sobre 2 grados polinómicos distintos, lo cual se ve a continuación:

Código 10: Grilla SVM kernel polinomial

```
tiempo_svmpolinomial1_inicio = time.time()


parametrosp= {'degree':[2,3],'C':[0.1, 0.25, 0.75, 1],'gamma':['scale','auto',1/(len(entrenamiento)**2)
    ↪ ,1]}
svmpolinomial1 = svm.SVC(kernel= 'poly', probability=False)
clfp1 = GridSearchCV(svmpolinomial, parametrosp, cv=5)
clfp1.fit(scaled_entrenamiento,entrenamiento.iloc[:,10])

tiempo_svmpolinomial1_final = time.time()
print(clfp1.best_params_)


########Entrenamiento segundo svm polinomial########
tiempo_svmpolinomial2_inicio = time.time()
svmpolinomial2 = svm.SVC(kernel= 'poly', probability=False, degree= 2, gamma= 'scale')
svmpolinomial2.fit(scaled_entrenamiento,entrenamiento.iloc[:,10])
tiempo_svmpolinomial2_final = time.time()
```

Donde *clfl* tiene guardado al SVM con kernel polinomial con la mejor combinación de hiper parámetros dados. La función *best_params_* retorna los mejores parámetros de la grilla, en este caso 'C': 0.25, 'degree': 3, 'gamma': 1. Por otra parte, dado que posteriormente se analizará la comparación entre el mejor SVM de kernel polinomial con los parámetros establecidos, se genera un segundo SVM el cual no pasa por una grilla, con los parámetros 'C': 1, 'degree': 2, 'gamma': 'scale' los cuales son los parámetros por defecto para este SVM a excepción del grado.

Posteriormente se procede igual que con SVM lineal. Para la evaluación de hiper parámetros se utiliza como test el conjunto de validación, del cual se obtiene la matriz de confusión. Esto se realiza mediante el código:

Código 11: Matriz de confusión SVM kernel polinomial

```
print('Mejor SVM polinomial')
y_pred_svmp1=clfp1.predict(scaled_validacion)
#sklearn.metrics.confusion_matrix(y_true, y_pred, *, labels=None, sample_weight=None,
    ↪ normalize=None)
print(sklearn.metrics.confusion_matrix(validacion.iloc[:,10], y_pred_svmp1, normalize=None))
print(sklearn.metrics.confusion_matrix(validacion.iloc[:,10], y_pred_svmp1, normalize='true'))

##########segundo svm polinomial
print('Otro SVM polinomial')
```

```
y_pred_svmp2=svmpolinomial2.predict(scaled_validacion)
print(sklearn.metrics.confusion_matrix(validacion.iloc[:,10], y_pred_svmp2, normalize=None))
print(sklearn.metrics.confusion_matrix(validacion.iloc[:,10], y_pred_svmp2, normalize='true'))
```

La cual retorna las siguientes matrices de confusión:

Tabla 4: Matriz de confusión SVM kernel polinomial con mejores parámetros de grilla

	Negativos Predichos	Positivos Predichos
Negativos Observados	491	96
Positivos Observados	121	492

Tabla 5: Matriz de confusión SVM kernel polinomial con otros parámetros

	Negativos Predichos	Positivos Predichos
Negativos Observados	504	83
Positivos Observados	179	434

Las cuales normalizadas retornan los siguientes parámetros:

Tabla 6: Matriz de confusión normalizada SVM kernel polinomial con mejores parámetros de grilla

	Negativos Predichos	Positivos Predichos
Negativos Observados	0.84	0.16
Positivos Observados	0.2	0.8

Posteriormente se genera una curva ROC para mostrar el desempeño de los clasificadores, esto mediante el código:

Código 12: Curva ROC SVM de kernel polinomial

```

y_score_svmp1=clfp1.decision_function(scaled_validacion)
fprp1, tprp1, thresholds = sklearn.metrics.roc_curve(validacion.iloc[:,10], y_score_svmp1)
area_bajo_curva_svmp1=metrics.auc(fprp1, tprp1)

plt.figure()

plt.plot(fprp1, tprp1, color='red', label='Curva ROC SVMP1 (area = %0.2f)' %
    ↪ area_bajo_curva_svmp1)
plt.plot([0, 1], [0, 1], color='blue',  linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Curva ROC SVM Polinomial con mejores parámetros para validación')
plt.legend(loc="lower right")
```

Tabla 7: Matriz de confusión normalizada SVM kernel polinomial con otros parámetros

	Negativos Predichos	Positivos Predichos
Negativos Observados	0.86	0.14
Positivos Observados	0.29	0.71

```
plt.show()

##########segundo svm polinomial

y_score_svmp2=svmpolinomial2.decision_function(scaled_validacion)
fprp2, tprp2, thresholds = sklearn.metrics.roc_curve(validacion.iloc[:,10], y_score_svmp2)
area_bajo_curva_svmp2=metrics.auc(fprp2, tprp2)

plt.figure()

plt.plot(fprp2, tprp2, color='red', label='Curva ROC SVMP2 (area = %0.2f)' %
    ↪ area_bajo_curva_svmp2)
plt.plot([0, 1], [0, 1], color='blue', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Curva ROC segundo SVM Polinomial para validación ')
plt.legend(loc="lower right")
plt.show()
```

El cual genera las siguientes curvas ROC:

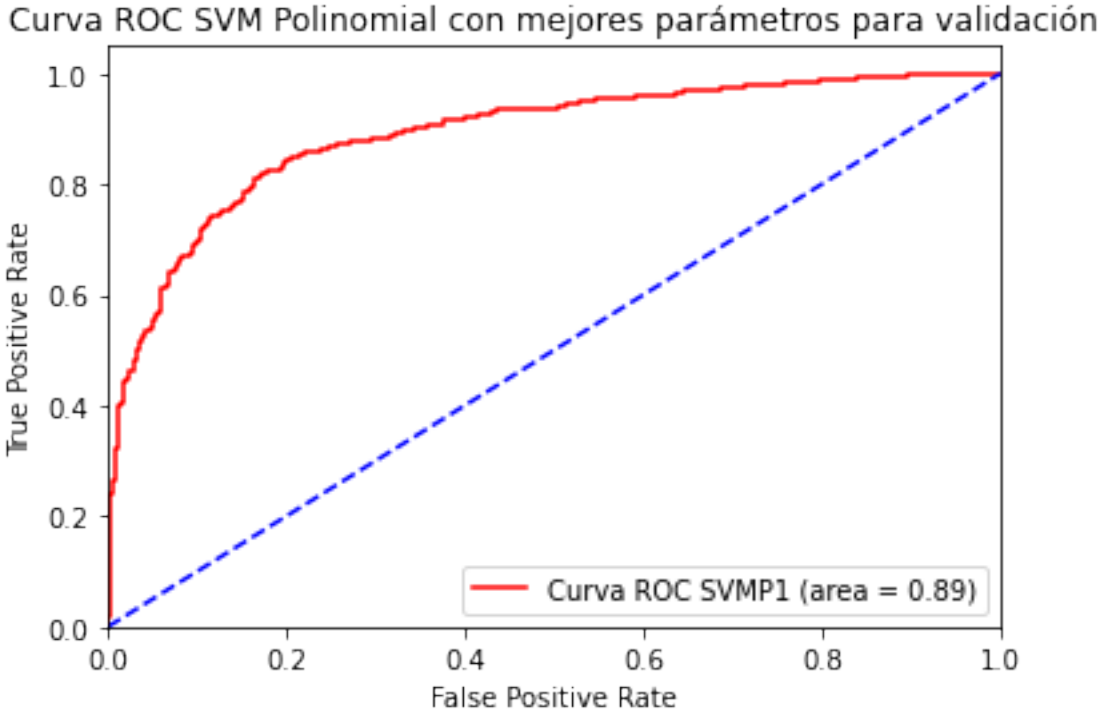

Figura 3: Curva ROC SVM kernel polinomial con mejores parámetros de grilla

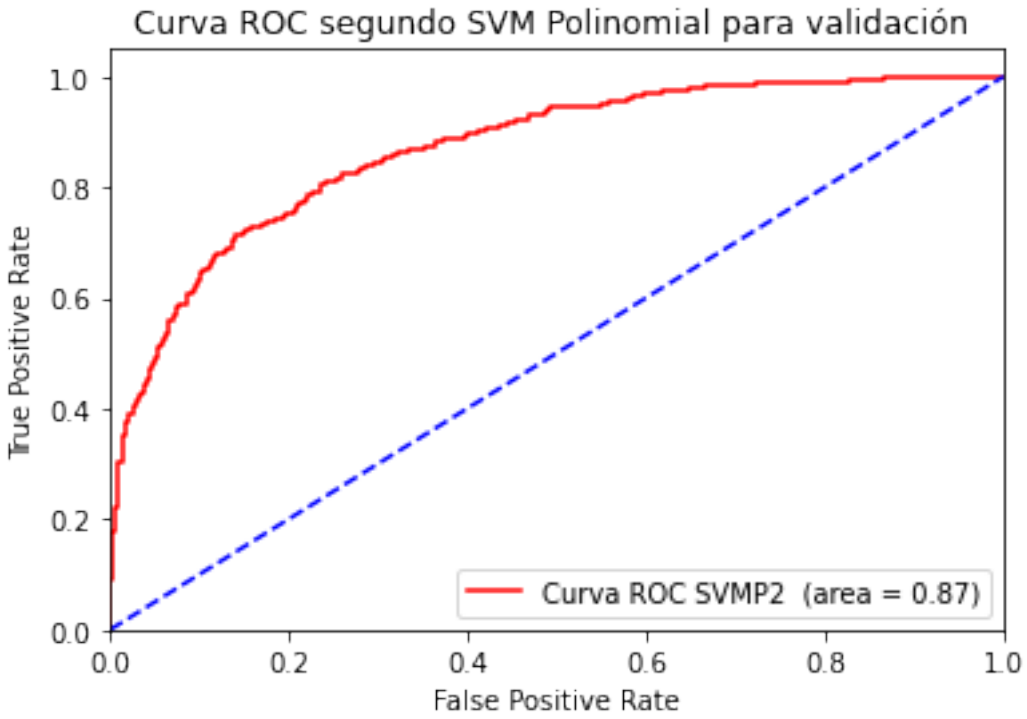

Figura 4: Curva ROC SVM kernel polinomial con otros parámetros

De las cuales podemos ver que el área bajo la curva del kernel con mejores parámetros de la grilla es 0.89, mientras que para el segundo SVM de kernel polinomial su área es de 0.87.

Por último se deben obtener sus respectivas curva precision recall, siendo el código para implementarlas el siguiente:

Código 13: Precision recall SVM kernel polinomial

```
1
2  precisionp, recallp, thresholds = sklearn.metrics.precision_recall_curve(validacion.iloc[:,10],
       ↪ y_score_svmp)
3  avg_ps_svmp=sklearn.metrics.average_precision_score(validacion.iloc[:,10], y_score_svmp)
4
5  plt.figure()
6
7  plt.plot(recallp,precisionp, color='red', label='Precision Recall SVMP1 (average precision = %0.2f)'
       ↪ % avg_ps_svmp)
8  plt.plot([0, 1], [1,0], color='blue',  linestyle='--')
9  plt.xlim([0.0, 1.0])
10 plt.ylim([0.0, 1.05])
11 plt.xlabel('Recall')
12 plt.ylabel('Precision')
13 plt.title('Curva Precision Recall SVM Polinomial con mejores parámetros para validación')
14 plt.legend(loc="lower left")
15 plt.show()
16
17
18 ##############Segundo SVM polinomial
19
20 precisionp2, recallp2, thresholds = sklearn.metrics.precision_recall_curve(validacion.iloc[:,10],
       ↪ y_score_svmp2)
21 avg_ps_svmp2=sklearn.metrics.average_precision_score(validacion.iloc[:,10], y_score_svmp2)
22
23 plt.figure()
24
25 plt.plot(recallp2,precisionp2, color='red', label='Precision Recall SVMP2 (average precision = %0.2f)
       ↪ ' % avg_ps_svmp2)
26 plt.plot([0, 1], [1,0], color='blue',  linestyle='--')
27 plt.xlim([0.0, 1.0])
28 plt.ylim([0.0, 1.05])
29 plt.xlabel('Recall')
30 plt.ylabel('Precision')
31 plt.title('Curva Precision Recall segundo SVM Polinomial para validación')
32 plt.legend(loc="lower left")
33 plt.show()
```

El que retorna las siguientes gráficas:

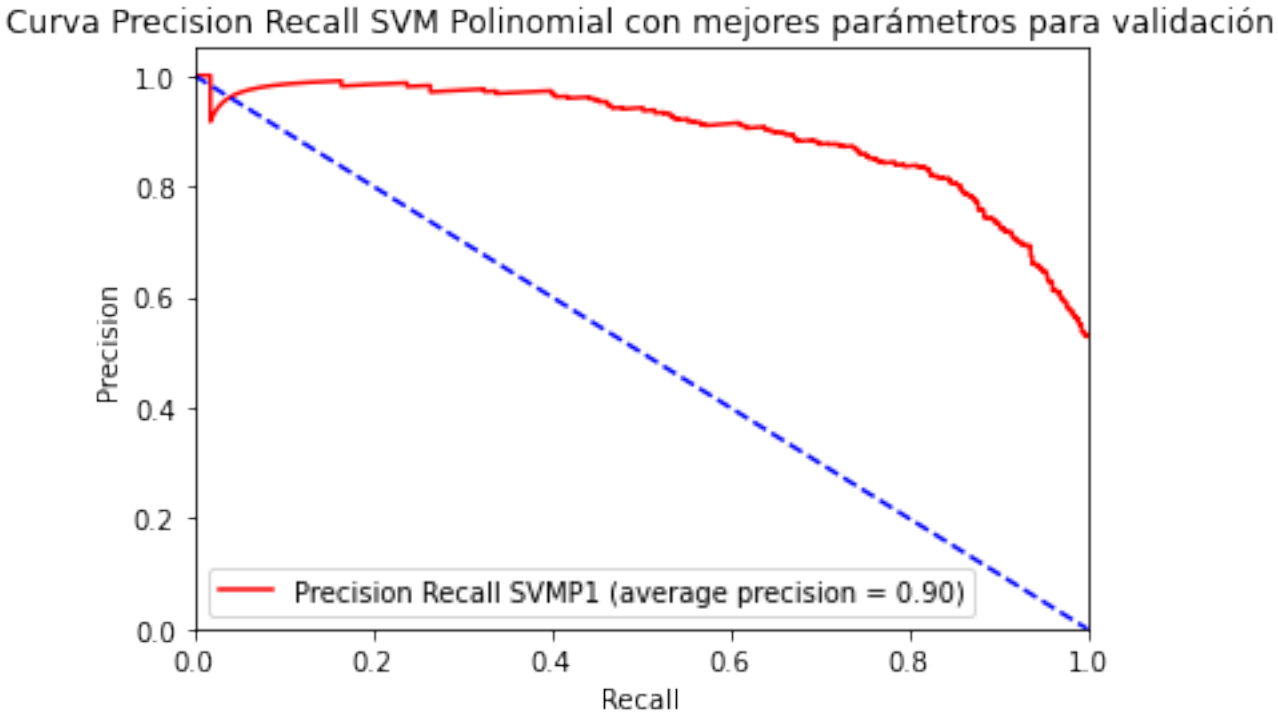

Figura 5: Curva precision recall SVM kernel polinomial con mejores parámetros de grilla

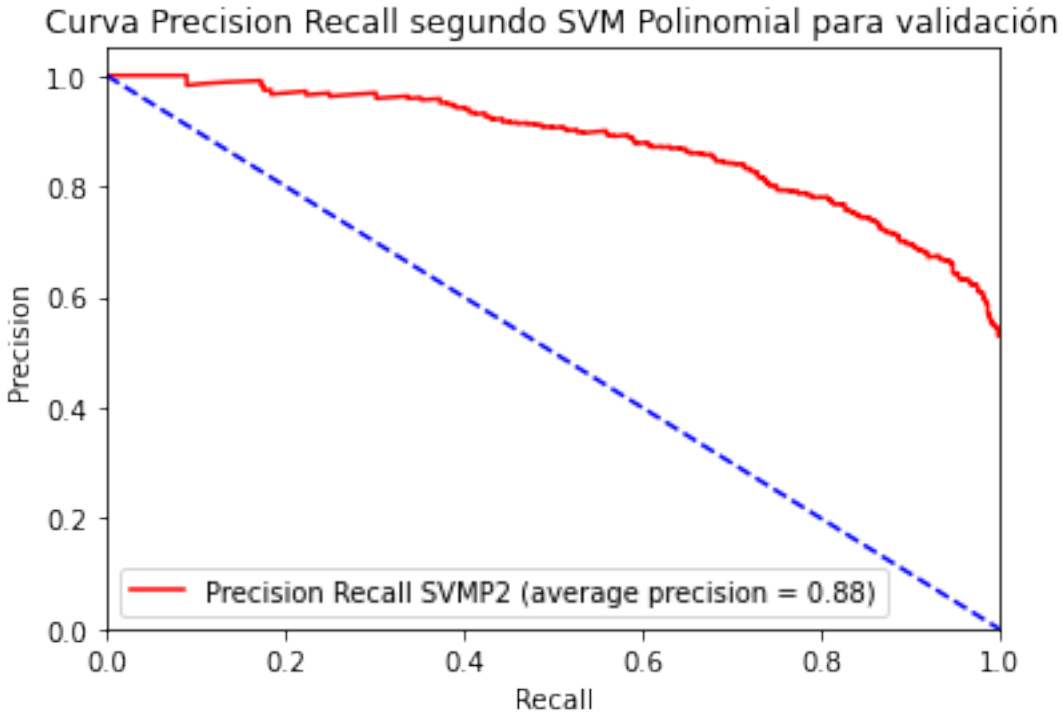

Figura 6: Curva precision recall SVM kernel polinomial con otros parámetros

Donde el average de SVM con mejores parámetros de grilla es 0.9, mientras que el segundo SVM posee un average de 0.88.

5. Parte 4: Clasificación usando SVM de kernel RBF y rendimiento

Para utilizar un SVM de kernel RBF lo primero es escoger hiper parámetros, razón por la cual se implementa una grilla, la cual consta de 5 valores para 'C' y 4 para 'Gamma', lo cual se ve a continuación:

Código 14: Grilla SVM kernel RBF

```
1
2 tiempo_svmrbf_inicio = time.time()
3
4
5 parametrosrbf= {'C':[0.1, 0.25,0.5, 0.75, 1],'gamma':['scale','auto',1/(len(entrenamiento)**2),1]}
6 svmrbf = svm.SVC(kernel= 'rbf', probability=False)
7 clfrbf = GridSearchCV(svmrbf, parametrosrbf, cv=5)
8 clfrbf.fit(scaled_entrenamiento,entrenamiento.iloc[:,10])
9 tiempo_svmrbf_final = time.time()
10 print(clfrbf.best_params_)
```

Donde *clfl* tiene guardado al SVM con kernel RBF con la mejor combinación de hiper parámetros dados. La función *best_params_* retorna los mejores parámetros de la grilla, en este caso 'C': 0.75, 'gamma': 'auto'.

Posteriormente se procede de igual manera que con los anteriores SVM. Para la evaluación de hiper parámetros se utiliza como test el conjunto de validación, del cual se obtiene la matriz de confusión:

Código 15: Matriz de confusión SVM kernel RBF

```
1
2 y_pred_svmrbf=clfrbf.predict(scaled_validacion)
3 print(sklearn.metrics.confusion_matrix(validacion.iloc[:,10], y_pred_svmrbf, normalize=None))
4 print(sklearn.metrics.confusion_matrix(validacion.iloc[:,10], y_pred_svmrbf, normalize='true'))
```

Que retorna las siguientes matrices:

Tabla 8: Matriz de confusión SVM kernel RBF

	Negativos Predichos	Positivos Predichos
Negativos Observados	507	80
Positivos Observados	122	491

O su versión normalizada:

Posteriormente se genera una curva ROC para mostrar el desempeño del clasificador mediante el siguiente código:

Tabla 9: Matriz de confusión normalizada SVM kernel RBF

	Negativos Predichos	Positivos Predichos
Negativos Observados	0.86	0.14
Positivos Observados	0.2	0.8

Código 16: Curva ROC SVM de kernel RBF

```
1
2  y_score_svmrbf=clfrbf.decision_function(scaled_validacion)
3  fprrbf, tprrbf, thresholds = sklearn.metrics.roc_curve(validacion.iloc[:,10], y_score_svmrbf)
4  area_bajo_curva_svmrbf=metrics.auc(fprrbf, tprrbf)
5
6  plt.figure()
7
8  plt.plot(fprrbf, tprrbf, color='red', label='Curva ROC SVMRBF (area = %0.2f)' %
       ↪ area_bajo_curva_svmrbf)
9  plt.plot([0, 1], [0, 1], color='blue',  linestyle='--')
10 plt.xlim([0.0, 1.0])
11 plt.ylim([0.0, 1.05])
12 plt.xlabel('False Positive Rate')
13 plt.ylabel('True Positive Rate')
14 plt.title('Curva ROC SVM RBF para validación')
15 plt.legend(loc="lower right")
16 plt.show()
```

El que genera la curva ROC:

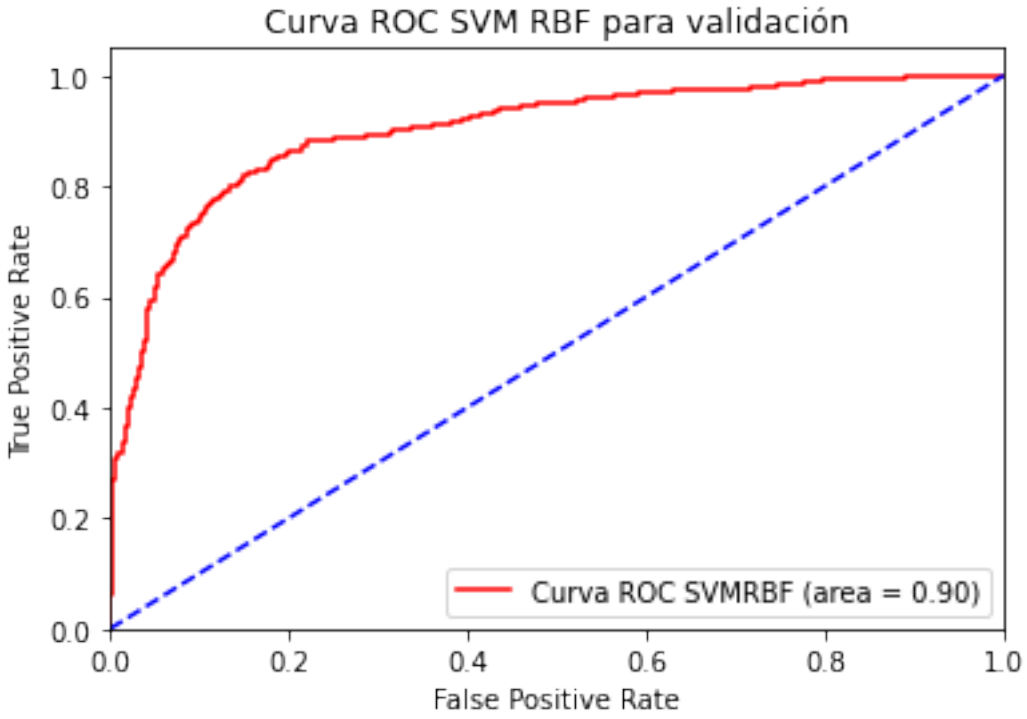

Figura 7: Curva ROC SVM kernel RBF

De la cual podemos observar que tiene un área bajo la curva de 0.9.

Por último para obtener su curva precision recall se genera el código:

Código 17: Curva precision recall SVM kernel RBF

```
1
2 precisionrbf, recallrbf, thresholdsrbf = sklearn.metrics.precision_recall_curve(validacion.iloc[:,10],
      ↪ y_score_svmrbf)
3 avg_ps_svmrbf=sklearn.metrics.average_precision_score(validacion.iloc[:,10], y_score_svmrbf)
4
5 plt.figure()
6
7 plt.plot(recallrbf,precisionrbf, color='red', label='Precision Recall SVMRBF (average precision =
      ↪ %0.2f)' % avg_ps_svmrbf)
8 plt.plot([0, 1], [1,0], color='blue',  linestyle='--')
9 plt.xlim([0.0, 1.0])
10 plt.ylim([0.0, 1.05])
11 plt.xlabel('Recall')
12 plt.ylabel('Precision')
13 plt.title('Curva Precision Recall SVM RBF para validación')
14 plt.legend(loc="lower left")
15 plt.show()
```

Retornando el siguiente gráfico:

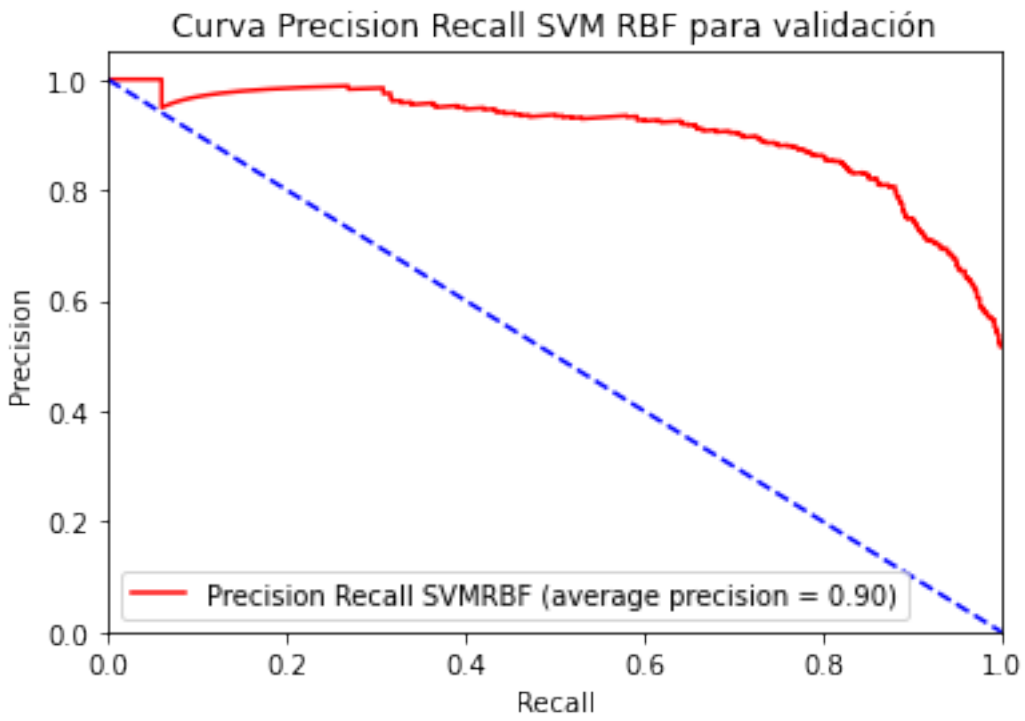

Figura 8: Curva precision recall SVM kernel RBF

Donde el average de este es 0.9.

5.1. Parte 5: Comparación de resultados

Se evalúa el mejor clasificador de cada tipo, incluyendo 2 SVM de kernel polinomial.
Lo primero es generar la matriz de confusión para cada SVM sobre el conjunto de pruebas, lo cual es generado mediante:

Código 18: Matriz de confusión SVMs en conjunto de pruebas

```
1
2  #lineal
3  print('Matriz de confusión SVM lineal en test')
4  y_pred_svml_test=clfl.predict(scaled_test)
5  print(sklearn.metrics.confusion_matrix(test.iloc[:,10], y_pred_svml_test, normalize=None))
6  print('Normalizada')
7  print(sklearn.metrics.confusion_matrix(test.iloc[:,10], y_pred_svml_test, normalize='true'))
8
9  #polinomios
10 print('Matriz de confusión SVM de mejor polinomio en test')
11 y_pred_svmp1_test=clfp1.predict(scaled_test)
12 print(sklearn.metrics.confusion_matrix(test.iloc[:,10], y_pred_svmp1_test, normalize=None))
13 print('Normalizada')
14 print(sklearn.metrics.confusion_matrix(test.iloc[:,10], y_pred_svmp1_test, normalize='true'))
15
16 ##########segundo svm polinomial
17 print('Matriz de confusión SVM de segundo polinomio en test')
18 y_pred_svmp2_test=svmpolinomial2.predict(scaled_test)
19 print(sklearn.metrics.confusion_matrix(test.iloc[:,10], y_pred_svmp2_test, normalize=None))
20 print('Normalizada')
21 print(sklearn.metrics.confusion_matrix(test.iloc[:,10], y_pred_svmp2_test, normalize='true'))
22
23 ##############rbf
24 print('Matriz de confusión SVM RBF en test')
25 y_pred_svmrbf_test=clfrbf.predict(scaled_test)
26 print(sklearn.metrics.confusion_matrix(test.iloc[:,10], y_pred_svmrbf_test, normalize=None))
27 print('Normalizada')
28 print(sklearn.metrics.confusion_matrix(test.iloc[:,10], y_pred_svmrbf_test, normalize='true'))
```

Generando las siguientes matrices de confusión normalizadas:

Tabla 10: Matriz de confusión normalizada SVM lineal sobre conjunto de pruebas

	Negativos Predichos	Positivos Predichos
Negativos Observados	0.79	0.21
Positivos Observados	0.25	0.75

Para comparar los resultados se utilizan gráficos de evaluación de rendimiento como son el caso de las curvas ROC y Precision-Recall.

Para el gráfico de curvas ROC superpuestas se utiliza el siguiente código:

Tabla 11: Matriz de confusión normalizada SVM kernel polinomial con mejores parámetros de grilla sobre conjunto de pruebas

	Negativos Predichos	Positivos Predichos
Negativos Observados	0.82	0.18
Positivos Observados	0.20	0.8

Tabla 12: Matriz de confusión normalizada SVM kernel polinomial con otros parámetros sobre conjunto de pruebas

	Negativos Predichos	Positivos Predichos
Negativos Observados	0.83	0.17
Positivos Observados	0.32	0.68

Código 19: Comparación de curvas ROC

```
1
2  #lineal
3  y_score_svml_test=clf.decision_function(scaled_test)
4  fprl_test, tprl_test, thresholds = sklearn.metrics.roc_curve(test.iloc[:,10], y_score_svml_test)
5  area_bajo_curva_svml_test=metrics.auc(fprl_test, tprl_test)
6
7  #mejor polinomio
8  y_score_svmp1_test=clfp1.decision_function(scaled_test)
9  fprp1_test, tprp1_test, thresholds = sklearn.metrics.roc_curve(test.iloc[:,10], y_score_svmp1_test)
10 area_bajo_curva_svmp1_test=metrics.auc(fprp1_test, tprp1_test)
11
12 #segundo svm polinomial
13 y_score_svmp2_test=svmpolinomial2.decision_function(scaled_test)
14 fprp2_test, tprp2_test, thresholds = sklearn.metrics.roc_curve(test.iloc[:,10], y_score_svmp2_test)
15 area_bajo_curva_svmp2_test=metrics.auc(fprp2_test, tprp2_test)
16
17 #RBF
18 y_score_svmrbf_test=clfrbf.decision_function(scaled_test)
19 fprrbf_test, tprrbf_test, thresholds = sklearn.metrics.roc_curve(test.iloc[:,10], y_score_svmrbf_test)
20 area_bajo_curva_svmrbf_test=metrics.auc(fprrbf_test, tprrbf_test)
21
22
23
24
25 plt.figure()
26
27 plt.plot(fprl_test, tprl_test, color='red', label='Curva ROC SVML (area = %0.2f)' %
       ↪ area_bajo_curva_svml_test)
28 plt.plot(fprp1_test, tprp1_test, color='green', label='Curva ROC SVMP1 (area = %0.2f)' %
       ↪ area_bajo_curva_svmp1_test)
29 plt.plot(fprp2_test, tprp2_test, color='purple', label='Curva ROC SVMP2  (area = %0.2f)' %
       ↪ area_bajo_curva_svmp2_test)
30 plt.plot(fprrbf_test, tprrbf_test, color='black', label='Curva ROC SVMRBF (area = %0.2f)' %
```

Tabla 13: Matriz de confusión normalizada SVM kernel RBF sobre conjunto de pruebas

	Negativos Predichos	Positivos Predichos
Negativos Observados	0.85	0.15
Positivos Observados	0.20	0.8

```
    ↪ area_bajo_curva_svmrbf_test)
plt.plot([0, 1], [0, 1], color='blue',  linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Curva ROC SVMs para test')
plt.legend(loc="lower right")
plt.show()
```

El que en primera instancia calcula los parámetros fpr y tpr para posteriormente graficarlos, como se ve en la siguiente figura:

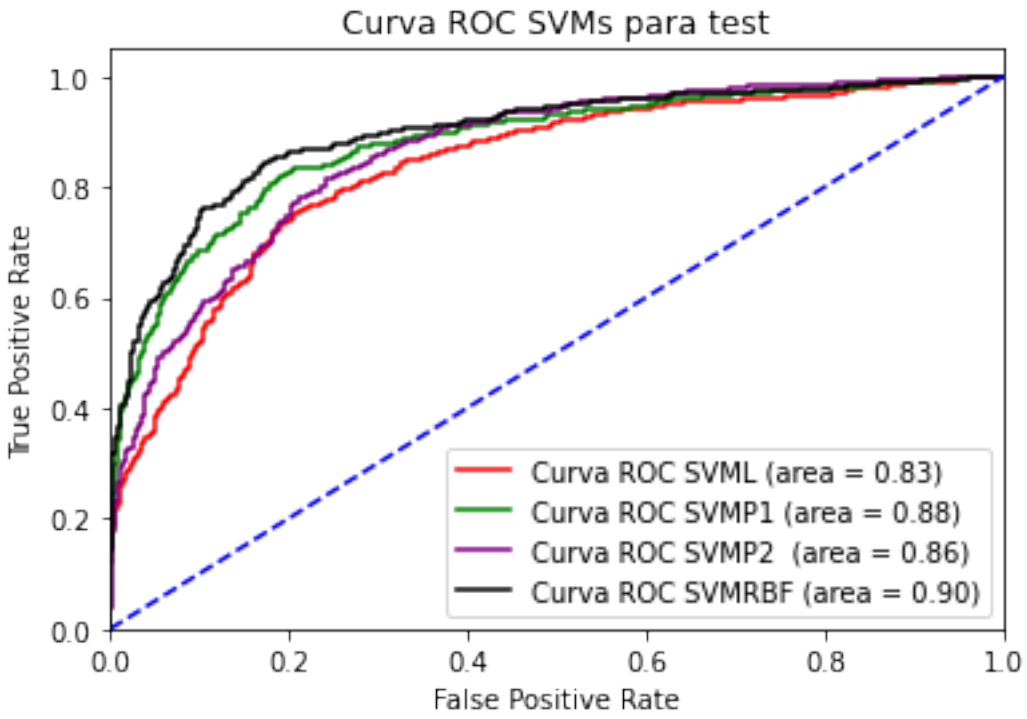

Figura 9: Comparación de curvas ROC

Donde podemos ver que el área bajo la curva es 0.83 para el SVM lineal, 0.88 para el SVM de kernel polinomial con los mejores parámetros de la grilla, 0.86 para el segundo SVM de kernel polinomial y 0.9 para el SVM RBF.

Para el gráfico de las curvas Precision-Recall superpuestas el código es el siguiente:

Código 20: Comparación de curvas precision recall

```

#lineal
precisionl_test, recalll_test, thresholds = sklearn.metrics.precision_recall_curve(test.iloc[:,10], y_score_svml_test)
avg_ps_svml_test=sklearn.metrics.average_precision_score(test.iloc[:,10], y_score_svml_test)

#mejor polinomio
precisionp1_test, recallp1_test, thresholds = sklearn.metrics.precision_recall_curve(test.iloc[:,10], y_score_svmp1_test)
avg_ps_svmp1_test=sklearn.metrics.average_precision_score(test.iloc[:,10], y_score_svmp1_test)

#segundo polinomio
precisionp2_test, recallp2_test, thresholds = sklearn.metrics.precision_recall_curve(test.iloc[:,10], y_score_svmp2_test)
avg_ps_svmp2_test=sklearn.metrics.average_precision_score(validacion.iloc[:,10], y_score_svmp2_test)

#RBF
precisionrbf_test, recallrbf_test, thresholdsrbf = sklearn.metrics.precision_recall_curve(test.iloc [:,10], y_score_svmrbf_test)
avg_ps_svmrbf_test=sklearn.metrics.average_precision_score(test.iloc[:,10], y_score_svmrbf_test)




plt.figure()

plt.plot(recalll_test,precisionl_test, color='red', label='Precision Recall SVML (average precision = %0.2f)' % avg_ps_svml_test)
plt.plot(recallp1_test,precisionp1_test, color='green', label='Precision Recall SVMP1 (average precision = %0.2f)' % avg_ps_svmp1_test)
plt.plot(recallp2_test,precisionp2_test, color='purple', label='Precision Recall SVMP2 (average precision = %0.2f)' % avg_ps_svmp2_test)
plt.plot(recallrbf_test,precisionrbf_test, color='black', label='Precision Recall SVMRBF (average precision = %0.2f)' % avg_ps_svmrbf_test)
plt.plot([0, 1], [1,0], color='blue',  linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Curva Precision Recall SVMs para test')
plt.legend(loc="lower left")
plt.show()
```

Este código genera la siguiente figura:

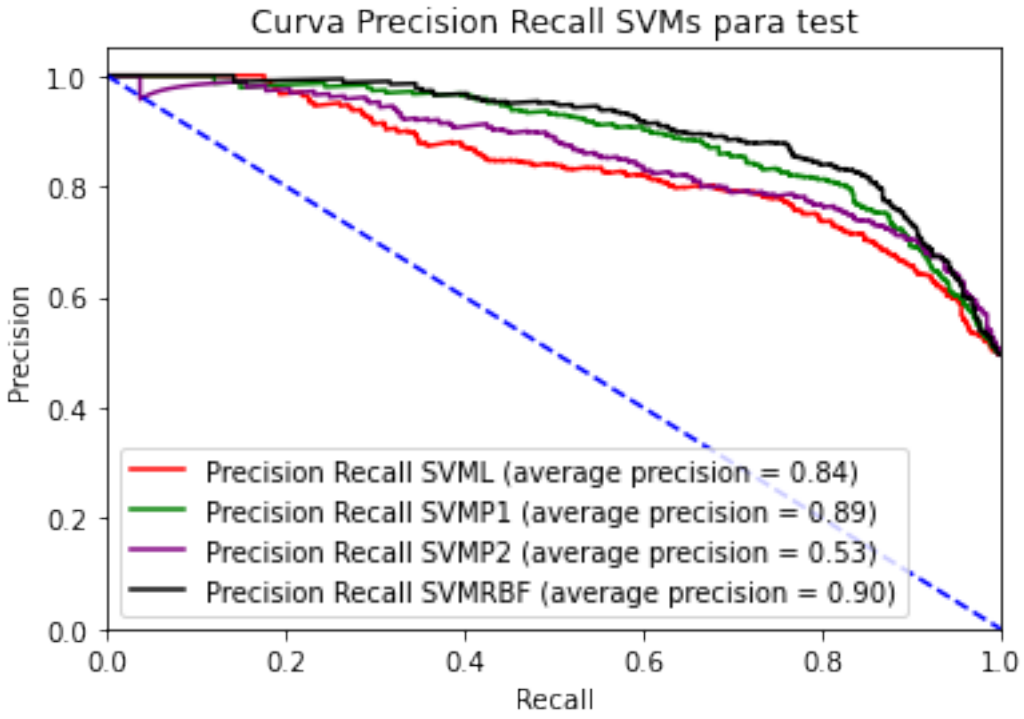

Figura 10: Comparación de curvas Precision-Recall

Donde el average de SVM lineal es de 0.84, mientras que para el de kernel polinomial de mejores parámetros de grilla es 0.89, el segundo polinomial tan solo posee 0.53 de average y el SVM de kernel RBF posee un average de 0.90.

El último punto de comparación son los tiempos de respuesta de cada entrenamiento, lo cual es visualizado con la funcion **time.time()**, la cual fue utilizada para medir los tiempos que requieren cada clasificador para ser entrenados. Los tiempos de ejecución son presentados a través del código:

Código 21: Comparación tiempos de ejecución

```
1
2 print('Tiempos de entrenamiento')
3 print('Tiempo SVM lineal: ',tiempo_svmlineal_final-tiempo_svmlineal_inicio,'[s]')
4 print('Tiempo mejor SVM polinomial: ',tiempo_svmpolinomial1_final-
    ↪ tiempo_svmpolinomial1_inicio,'[s]')
5 print('Tiempo SVM polinomial sin grillar: ',tiempo_svmpolinomial2_final-
    ↪ tiempo_svmpolinomial2_inicio,'[s]')
6 print('Tiempo SVM RBF: ',tiempo_svmrbf_final-tiempo_svmrbf_inicio,'[s]')
```

Dando como respuesta lo siguiente:

Tabla 14: Comparación de tiempos de entrenamiento

Tipo de SVM	Tiempo de entrenamiento[s]
Lineal	4.98
Kernel polinomial mejores parametros de grilla	93.10
Kernel polinomial parametros sin grilla	0.26
Kernel RBF	34.94

6. Conclusiones

Los resultados obtenidos por los diferentes SVM fueron bastante aceptables, aunque el mejor es el con kernel RBF ya que obtuvo la mejor calificación tanto en curva ROC como precision recall. Cabe destacar de igual forma que el tiempo de entrenamiento del SVM con kernel polinomial sin grillar sorprende bastante, debido a que su tiempo de entrenamiento fue el más bajo y aún así su curva ROC en el conjunto de testeo fue mejor que la de SVM lineal, aunque se puede deducir con la curva precision recall que este SVM puede ser propenso a sesgo.

Para mejorar el desempeño de los clasificadores podría ser una buena idea el generar variados conjuntos de validación, utilizar las mismas grillas y estudiar qué hiper parámetros se repiten más.

Como aprendizaje al llevar a cabo la experiencia se logró un acercamiento a la biblioteca sklearn.