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

Tarea 3

Support Vector Machine

Integrantes: Lukas Pavez
Profesor: Javier Ruiz del Solar
Auxiliar: Patricio Loncomilla
Ayudantes: Francisco Leiva C.
Gabriel Andrés Azócar Cárcamo
Giovanni Pais L.
Nicolas Cruz Brunet

Fecha de entrega: 1 de Mayo de 2019
Santiago, Chile

Índice de Contenidos

Índice de Figuras

1. Introducción

Se tiene como objetivo implementar un clasificador de revisiones de restaurantes. Se dispone de un set de datos entregados en la tarea que contiene 1000 revisiones, donde cada una tiene una etiqueta que indica si la revisión fue favorable (1) o desfavorable (0).

Para lograr esto se entrenará un clasificador Support Vector Machine (SVM). Se estudiará el desempeño del clasificador utilizando distintos tipos de kernel.
Se utilizará una grilla para encontrar los mejores hiperparámetros del clasificador, probando con distintos valores para C, esto se hará con el set de entrenamiento, luego se probará el desempeño del clasificador con distintos kernel con el set de validación, finalmente se clasificará el set de prueba utilizando el mejor kernel obtenido en la parte anterior.

2. Marco Teórico

En esta tarea se trabaja con support vector machines, que son un modelo de aprendizaje supervisado, donde se busca el mejor plano que separa las clases, el objetivo del clasificador es encontrar el plano que mejor separa las clases, esto es que el margen (o distancia) entre el plano y los puntos más cercanos al plano (también llamados vectores de soporte) sea máximo, luego de encontrar el plano, los puntos que caigan a un lado son de una clase y los puntos que caen al otro lado son de la otra clase. Para separar las clases con un plano, el set de datos debe ser linealmente separable, y se tiene la siguiente formulación para solucionar el problema:

$$\max_{\alpha_i} J(\alpha_i) = -\frac{1}{2}\sum_{i=1}^{NT}\sum_{j=1}^{NT} y_i y_j \alpha_i \alpha_j \delta_i^T \delta_j + \sum_{i=1}^{NT}\alpha_i$$

$$\text{sujeto a:} \quad \sum_{i=1}^{NT} y_i \alpha_i = 0$$

$$0 \le \alpha_i \le C \qquad \text{i} = 1,...,\text{NT}$$

Figura 1: Formulación caso lineal

Donde los y son las etiquetas, α son multiplicadores y δ son los vectores de soporte.

En el caso de que los datos no sean linealmente separables, se utiliza un kernel, que lo que hace es aumentar la dimensionalidad del problema para así encontrar el hiperplano que separa las clases, y se debe maximizar el margen de ese hiperplano, lo importante es que el kernel hace la transformación de forma implícita, ya que realmente no se calculan las coordenadas de los puntos en el espacio de mayor dimensionalidad. El kernel cambia el producto punto entre los vectores de soporte en la Figura 2 por la aplicación del kernel escogido a esos vectores, por lo que la formulación queda de la siguiente forma:

$$\max_{\alpha_i} J(\alpha_i) = -\frac{1}{2}\sum_{i=1}^{NT}\sum_{j=1}^{NT} y_i y_j \alpha_i \alpha_j \, K(\delta_i, \delta_j) + \sum_{i=1}^{NT} \alpha_i$$

$$\text{sujeto a:} \quad \sum_{i=1}^{NT} y_i \alpha_i = 0$$

$$0 \le \alpha_i \le C \qquad i = 1,...,NT$$

Figura 2: Formulación caso no lineal

Hay distintos tipos de kernel, en esta tarea se utilizan 3: lineal, polinomial y rbf.

- lineal: este kernel no le hace nada al producto ya que asume que el problema es linealmente separable.
  $K(x_i, x_j) = x_i \cdot x_j$.
- polinomial: este kernel permite separar los puntos mediante un plano polinomial de grado d.
  $K(x_i, x_j) = (x_i \cdot x_j + 1)^d$.
- rbf: aquí, el espacio inducido por el kernel es un espacio gaussiano, donde cada punto se transforma en una distribución de probabilidad, tiene un parámetro gamma, que define el "radio de influencia' de cada punto, donde a mayor gamma, menor es la influencia de cada punto.
  $K(x_i, x_j) = \exp\left\{\frac{-\|x_i, x_j\|^2}{\gamma}\right\}$

Además de los parámetros de cada kernel, también se tiene el parámetro C, su valor indica al clasificador cuanto se quiere evitar errores de clasificación, para grandes valores de C, se buscarán margenes más pequeños para el hiperplano, mientras que para valores pequeños de C, los margenes serán mayores.

Para clasificar correctamente las revisiones de restaurantes, se utilizará el modelo Bag of Words, que es una representación simplificada de oraciones dentro de un texto, donde se tiene en cuenta la multiplicidad de cada palabra dentro de la oración. Para crear la bag of words del problema primero se crea una lista con todas las palabras dentro del dataset, y luego, para cada revisión, se crea otra lista, donde cada elemento es la cuenta de las palabras de la primera lista dentro de la revisión, y al hacerlo con todas las revisiones se tiene la matriz de características del set de datos.

En esta tarea se utilizará cross-validation, que es un método para evaluar que los datos de pas particiones de entrenamiento y prueba sean independientes, y se utiliza para evaluar la precisión del modelo a utilizar. Para llevar a cabo el proceso se necesita definir el número k de folds o iteraciones a utilizar, esto es ya que en el proceso, todo el conjunto se divide en k particiones, y para cada una de esas particiones, se toma una y se deja como conjunto de prueba y el resto se usa como conjunto de entrenamiento, y se repite para todas las particiones, por lo que se hacen k iteraciones, para obtener el resultado se obtiene la media aritmética de los resultados de las iteraciones.

3. Desarrollo

3.1. Lectura del dataset

Para leer el dataset se creó la función load_file(), esta función abre el archivo con la función open(), luego se salta una linea, ya que la primera linea del archivo no es una revisión, sino es una descripción de las columnas, por lo que esa linea no es relevante. Luego, para todas las lineas, separa la revisión de la etiqueta con la función split, y agrega la tupla revisión-etiqueta a una lista, al finalizar esto se entrega una matriz de 2 columnas, una son las revisiones y la otra son las etiquetas. El código es el siguiente:

```
1 def load_file():
2     file = open("Restaurant_Reviews_pl.tsv")
3     ar = []
4     file.readline()  # read 'Review Liked' line
5     for line in file:
6         text, label = line.split('\t')
7         ar.append([text, int(label)])
8     n_ar = np.array(ar)
9     return n_ar
```

3.2. Implementación BoW y división base de datos

Para crear la representación BoW de los datos simplemente se creo un objeto CountVectorizer() y se llamó al método fit_transform(datos), entregándole la columna de revisiones obtenida de la función de cargar los datos.

Para dividir la base de datos, primero se creo una matriz features que es la representación en matriz de la BoW obtenida, luego una lista labels que son las etiquetas transformadas de string a enteros, y finalmente se juntaron en una sola matriz features_with_labels, para esto se le agrego la columna labels a la matriz features con la función hstack de numpy, luego se llamó a la función split_data, que recibe la matriz con los datos y entrega el set de entrenamiento, validación y prueba. La función split_data parte por crear listas vacías good y bad, y para cada una de las filas de la matriz de datos, si la clase es 1 la agrega a good y si es 0 la agrega a bad, luego calcula los tamaños de los conjuntos, 60 % entrenamiento y 20 % validación, en el código aparece 80 % validación ya que se toma desde el tamaño de entrenamiento hasta el tamaño de validación (desde el 60 % hasta el 80 %). El resto se queda para el conjunto de prueba (20 %), se le hace esto a las 2 listas (good y bad), y después de separarlas, se combinan las listas respectivas de las 2 divisiones (por ejemplo entrenamiento good y entrenamiento bad), luego de combinarlas se les hace un shuffle a los 3 conjuntos y luego se retornan. El código de la implementación de BoW y la función split_data es el siguiente:

```python
if __name__ == "__main__":
    data = load_file()
    vectorizer = CountVectorizer()
    bow = vectorizer.fit_transform(data[:, 0])

    features = np.array(bow.toarray())
    labels = np.array(data[:, 1])
    labels = labels.astype(np.int)
    features_with_labels = np.hstack((features, labels.reshape(1000, 1)))

    train_set, valid_set, test_set = split_data(features_with_labels)

    ...

def split_data(data):
    s = len(data[0])
    good = []
    bad = []
    for x in data:
        if x[s-1] == 1:
            good.append(x)
        else:
            bad.append(x)

    good = np.array(good)
    bad = np.array(bad)
    train_good_len = len(good) * 60 // 100
    valid_good_len = len(good) * 80 // 100
    train_bad_len = len(bad) * 60 // 100
    valid_bad_len = len(bad) * 80 // 100

    # separate between train, validation and test set for each class (60-20-20)
    train_good = good[:train_good_len]
    valid_good = good[train_good_len:valid_good_len]
    test_good = good[valid_good_len:]

    train_bad = bad[:train_bad_len]
    valid_bad = bad[train_bad_len:valid_bad_len]
    test_bad = bad[valid_bad_len:]

    train_set = np.vstack((train_good, train_bad))  # combine the sets of each train set into one
     ↪ train set
    valid_set = np.vstack((valid_good, valid_bad))  # combine the sets of each val set into one val
     ↪ set
    test_set = np.vstack((test_good, test_bad))  # combine the sets of each test set into one test set

    np.random.shuffle(train_set)
    np.random.shuffle(valid_set)
    np.random.shuffle(test_set)

    return train_set, valid_set, test_set
```

3.3. Entrenamiento y evaluación

Para esta parte, se creó la función grid_and_roc(), que recibe un clasificador base svc, el conjunto de entrenamiento train y el conjunto a evaluar data. Esta función parte separando los conjuntos recibidos en características (x_data) y etiquetas (y_data), después se definen distintos valores para C, donde se escogió 1, 10, 100 y 1000 (se escogieron estos valores ya que el trabajo se baso en los ejemplos de la documentación de sklearn, [link]). Luego se crea la grilla con el objeto GridSearchCV(), dandole el clasificador base y los parámetros definidos, y seteando el número de folds en 5, para después llamar a la función fit de la grilla con el set de entrenamiento. Al terminar el fit, se obtiene el mejor clasificador encontrado en la grilla, que se obtiene en best_estimator_.

Después de encontrar el clasificador, se evalúa en el conjunto entregado llamando a la función predict(), se calcula la matriz de confusión con la función confusion_matrix() entregándole las etiquetas correctas y las predicciones, y después se calcula la curva ROC junto con el área bajo la curva, a la función roc_curve se le entregan las etiquetas correctas y el resultado de evaluar la función decision_function en los datos, y se obtiene los FPR y TPR, estas 2 listas se le entregan a la función auc() para calcular el área bajo la curva. Finalmente se hace un print a la matriz de confusión y se grafica la curva ROC. La función retorna el parámetro svc entregado al principio y el área bajo la curva calculada. El código de la función es el siguiente:

```
1  def split_text_and_labels(data):
2      s = len(data[0])
3      return data[:, :s-1], data[:, s-1]
4
5
6  def grid_and_roc(svc, train, data):
7      x_train, y_train = split_text_and_labels(train)
8      x_data, y_data= split_text_and_labels(data)
9
10     t1 = time.time()
11     parameters = {'C': [1, 10, 100, 1000]}
12     grid = GridSearchCV(svc, parameters, cv=5, n_jobs=-1)
13     grid.fit(x_train, y_train)
14
15     classifier = grid.best_estimator_
16     predictions = classifier.predict(x_data)
17     y_valid_score = classifier.decision_function(x_data)
18     print("-----------------------------------------------")
19     print(metrics.confusion_matrix(y_data, predictions))
20
21     fpr, tpr, _ = metrics.roc_curve(y_data, y_valid_score)
22     auc = metrics.auc(fpr, tpr)
23
24     kernel = svc.kernel
25     if kernel == 'linear':
26         label = 'kernel linear, auc = {:.3f}'.format(auc)
27     elif kernel == 'poly':
28         label = 'kernel polinomial, degree = {}, auc = {:.3f}'.format(svc.degree, auc)
29     else:
```

```
30        label = 'kernel rbf, auc = {:.3f}'.format(auc)
31    plt.plot(fpr, tpr, label=label)
32    t2 = time.time()
33    print(label)
34    print("training time = {:.2f}".format(t2 - t1))
35    print("-----------------------------------------------")
36    return svc, auc
```

3.4. Evaluación con distintos kernels

Para evaluar los distintos kernels se crearon 4 clasificadores base con la función svm.SVC(), dándole un kernel distinto a cada uno. Los kernels utilizados son: lineal, polinomial de grado 2, polinomial de grado 3 y rbf. Después de crearlos, para cada uno se llamó a la función grid_and_roc(), donde se va guardando el mayor área bajo la curva junto con el mejor clasificador. Finalmente, se grafican y la figura se guarda en la carpeta del programa, con el nombre de *kernel_comparation*. La figura obtenida y las matrices de confusión son las siguientes:

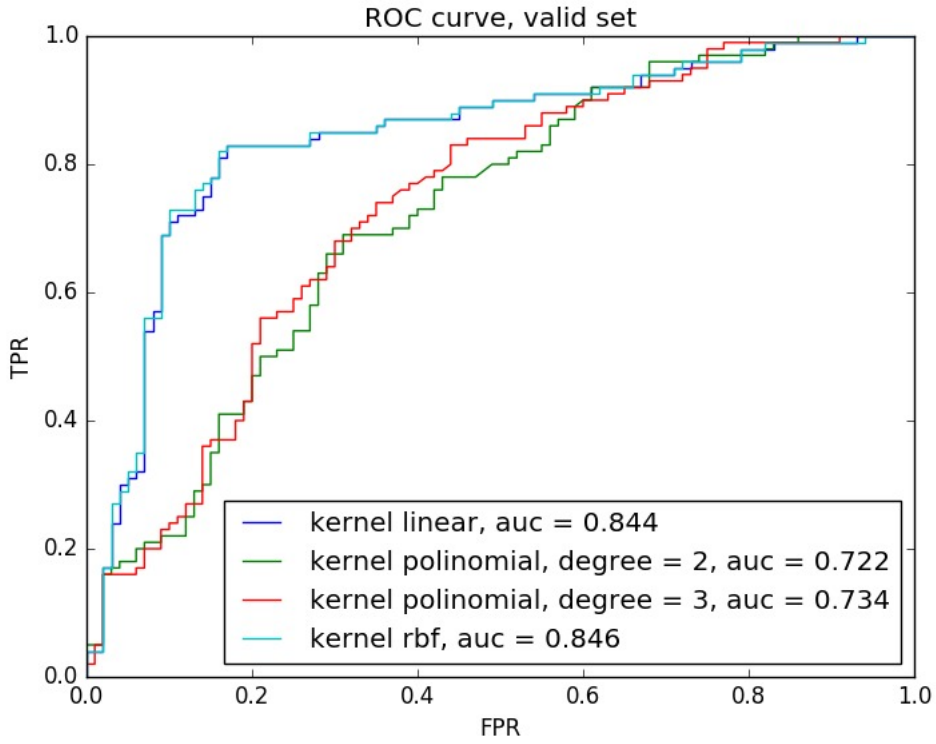

Figura 3: comparación de kernels

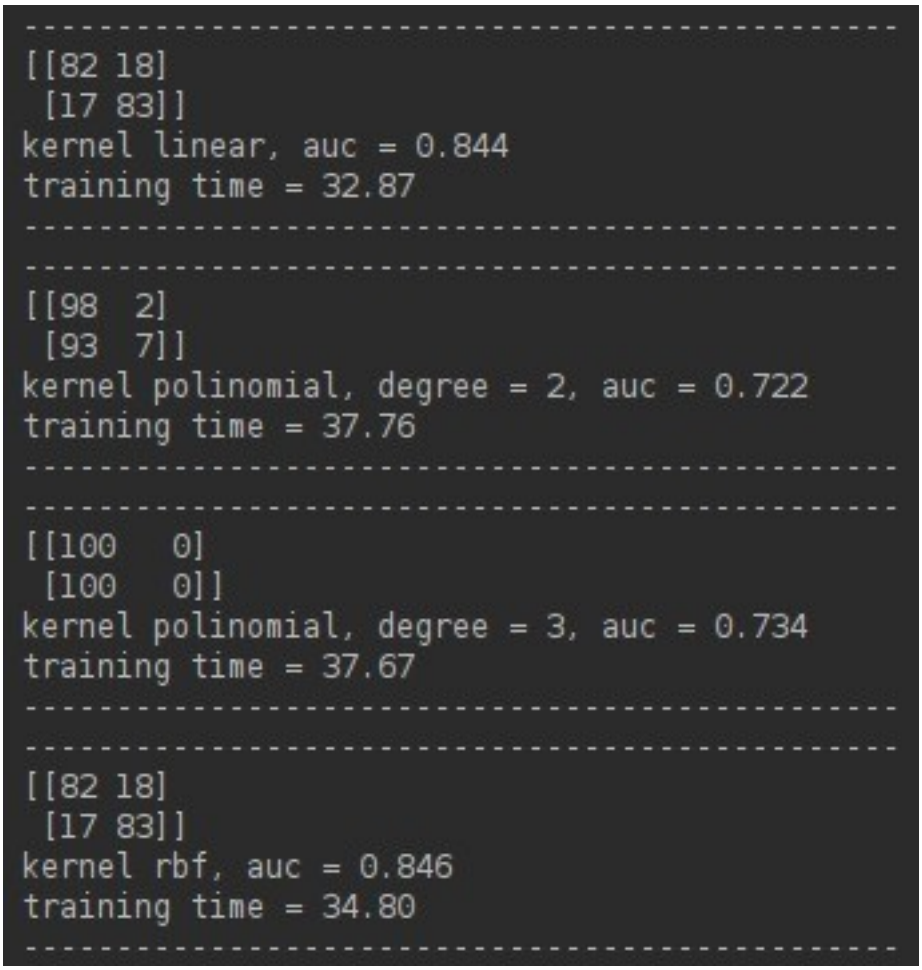

Figura 4: matrices de confusión de cada curva ROC

Luego de encontrar el mejor kernel, se vuelve a llamar a la función grid_and_roc dándole el mejor clasificador base encontrado y el conjunto de prueba, el gráfico obtenido se guarda con el nombre *best_kernel*. La figura obtenida y la matriz de confusión son las siguientes:

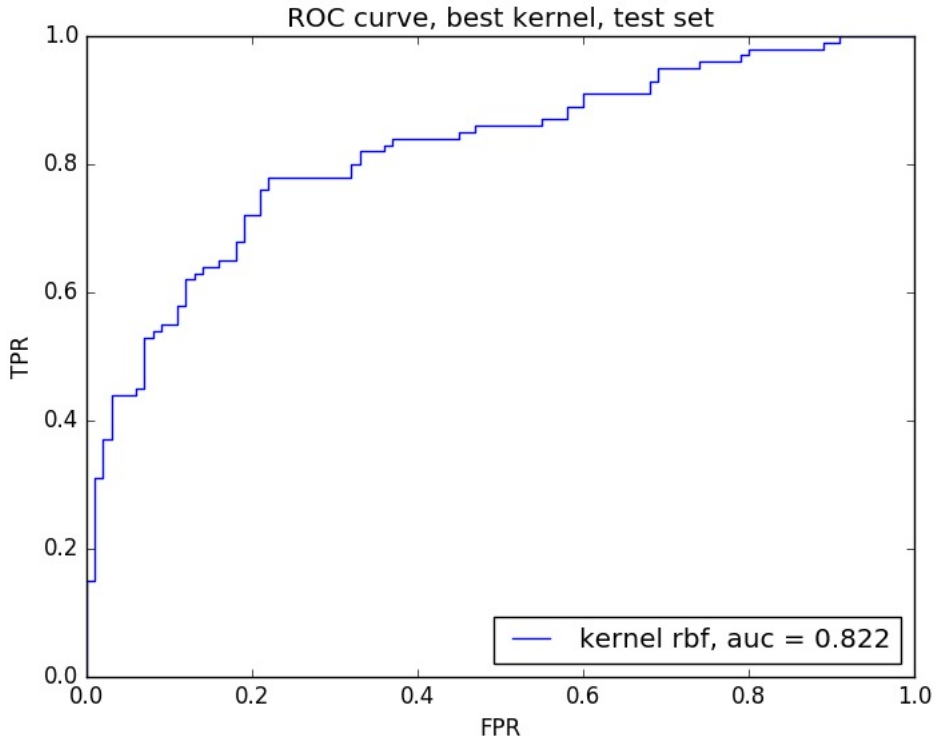

Figura 5: mejor kernel encontrado

```
---------------------------
[[70 30]
 [22 78]]
kernel rbf, auc = 0.822
training time = 34.86
---------------------------
```

Figura 6: matrices de confusión del mejor kernel encontrado

4. Conclusiones y Análisis de Resultados

Al ver los resultados de las comparaciones de las áreas bajo la curva de las ROC generadas, se pudo ver que las áreas producidas por los kernels lineal y rbf son muy parecidas, al correr distintas veces el programa una de las 2 aparecía con la mayor área, esto puede decir que al ser el problema es linealmente separable y no es necesario usar un kernel más sofisticado para resolver el problema.

Sobre los kernels polinomiales, por lo general entregaron valores bajos de área bajo la curva, se probó aumentar el grado, y se pudo ver que el área disminuía, por lo que se llegó a la conclusión de que el kernel polinomial no ayuda a resolver este problema.

También se pudo ver que el tiempo de ejecución es bajo, la mayoría de las curvas se demoraron aproximadamente 40 segundos, por lo que el tiempo no fue un factor importante al realizar la tarea.