Aprendizaje Automático (2018-2019)
Grado en Ingeniería Informática
Universidad de Granada

Proyecto final: Internet Advertisements Data Set

Montserrat Rodríguez Zamorano

7 de junio de 2019

Índice

1 Definición del problema a resolver.

Nuestro objetivo en este proyecto es proporcionar un modelo que sea capaz de predecir con éxito si una imagen determinada es un anuncio o no. Se trata, por tanto, de un problema de **clasificación**.

Se nos proporciona una base de datos que se puede encontrar en el enlace disponible en la bibliografía [1]. Este *dataset* representa un conjunto de imágenes, anuncios o no, en páginas de Internet.

La base de datos cuenta con un total de 3279 instancias y 1558 atributos. Se incluyen características tales como alto, ancho, o la URL de la imagen. En éstas, encontraremos tres variables continuas y el resto serán discretas. Más detalles acerca de los atributos pueden consultarse en el archivo *ad.names*.

Se consideraran los siguientes modelos:

- Regresión logística.
- Máquina de Soporte de Vectores (SVM).
- Random Forest.

2 Separación en conjuntos de train y test

Los datos se nos dan en un único archivo, sin separación en train y test ni atributos y etiquetas, por lo que tendremos que hacer esta separación nosotros mismos. En primer lugar separamos los atributos de las etiquetas. Si se observan los datos se puede comprobar la última columna es la que indica si se trata de un anuncio o no, por lo que separaremos esta última (1).

```
def separateLabels(data):
    array_data = data.values
    #guardamos los atributos
    x = np.array(array_data[:,0:1558], np.float64)
    #guardamos las etiquetas
    y = np.array(array_data[:,1558])
    return x,y
```

Listing 1: Código función para la separación en conjuntos de `train` y `test`

A continuación se separan los conjuntos train y test. Se reservará un 20 % para test, aunque la función (2) nos permite cambiar el porcentaje en cada algoritmo a través del parámetro `testsize`.

```
def setTrainTest(x,y, testsize):
    X_train, X_test, y_train, y_test = train_test_split(x, y,
        stratify=y, test_size=testsize, random_state=0)
    return X_train, X_test, y_train, y_test
```

Listing 2: Código función para la separación en conjuntos de `train` y `test`

Se fijará `random_state` para que los resultados sean los mismos en las ejecuciones y evitar así que los resultados que se muestran en la memoria sean distintos a una ejecución distinta que la que se hizo para su redacción.

3 Procesamiento de los datos de entrada

3.1. Tratamiento de datos perdidos

Como en cualquier problema real, nos encontramos ante la situación de tener que lidiar con datos perdidos o desconocidos. Vamos a encontrarnos con valores perdidos dentro de las variables continuas hasta en un 28 % de los casos. En esta base de datos, un dato perdido es aquel que se representa con *?*.

La función que puede verse en (3) se encarga de leer los datos y convertir todos los datos perdidos en no asignados.

```
def loadData(filedata):
    print('Loading␣data␣from␣file␣', filedata, '...')
    #convertir las ? en no  asignados
    cleandata = pd.read_csv(filedata, header =None, na_values=['␣␣␣
       ?', '␣␣␣␣␣?', '?'])
    print('Done')

    return cleandata
```

Listing 3: Código función para cargar datos

Una vez hecho esto se consideran distintas opciones para que estos datos perdidos no supongan un problema [4]:

- **Eliminar las instancias con valores faltantes.** Esto, sin embargo, nos llevaría a perder prácticamente un 30 % de los datos, lo que supone una pérdida que no podemos permitirnos.
- **Trabajar con los datos perdidos.** A veces, el hecho de que falte un dato puede proporcionarnos información. Sin embargo, hay modelos que no nos permiten trabajar con datos perdidos.
- **Sustituir estos datos por otros.** Por ejemplo, por la media de los valores conocidos de la columna en la que se encuentra.

Se trabajará con la tercera opción: se sustituirán los valores desconocidos por la media de los valores de la columna (4).

```
def convertMissingValues(data):
    #sustituir valores nan por la media de los valores de la
       columna
    imp = Imputer(missing_values=np.nan, strategy='mean')
    imp.fit(data)
    impdata = imp.transform(data)
    return impdata
```

Listing 4: Código para el tratamiento de datos perdidos

4 Justificación de la función de pérdida usada

Al tratarse de un problema de clasificación, la función de pérdida que se utilizará será la tasa de elementos clasificados erróneamente. Considero que es la función adecuada ya que no se puede clasificar bien a medias, o acercarse mucho a la clasificación (como pasaría en el caso de la regresión), sólo se puede clasificar bien o mal. Por tanto, contabilizar los aciertos y los fallos me parece una aproximación adecuada.

5 Regresión logística

En muchos casos los modelos lineales dan mejores resultados que alternativas más complicadas, por lo que es bueno comenzar por un modelo lineal.

Al tratarse de un problema de clasificación, un modelo lineal que puede usarse es *Regresión logística*.

Tal y como se indica en la lectura recomendada, se construirá un modelo rápido, se verán sus deficiencias y se intentarán solventar.

5.1. Selección de características importantes

La alta dimensionalidad de los datos puede resultar un problema. Seleccionar un conjunto más pequeño de características puede ayudar a evitar el sobreajuste. El método que utilizaremos será PCA (*Principal Componet Analysis*).

Es importante tener en cuenta que antes de seleccionar un subconjunto de características, tendremos que procesar los datos adecuadamente. [5] [6] [7]

5.1.1. Estandarización de datos

Estandarizar los datos consiste en escalar y centrar los datos de forma que la media será 0 y la varianza unidad. Matemáticamente, esto se hará restando la media y dividiendo entre la desviación estándar.

Como se indicó anteriormente, dentro de las características tenemos tres variables continuas y el resto discretas, con valores 0 o 1. Por tanto, la varianza será necesariamente mayor en el primer caso, dando más importancia a estas variables.[5] [6] [7]

Veamos a continuación qué pasaría si aplicáramos PCA sin estandarización.

```
X_train shape before feature selection: (2623, 1558)
X_train shape after feature selection: (2623, 2)

    Prediction without standarization
    Accuracy: 91.76829 %
    Eout: 0.08232
```

Figura 5.1: PCA sin estandarizar los datos

Como puede verse en (5.1), aunque el porcentaje de aciertos al predecir con los datos de test no sea demasiado bajo, estando ligeramente por encima del 90 %, estamos sesgando demasiado nuestro conjunto de características, pasando de 1558 características a tan solo 2.

A la hora de programar, es importante ajustar con datos de training y utilizar estos mismos parámetros en test. [7]

Se tienen en cuenta los siguientes parámetros:

- `with_mean`: tiene que ser `True` para que se reste la media.
- `with_std`: tiene que ser `True` para que se divida entre la desviación estándar.

Ambos serán `True` por defecto, por lo que no se especifica en el código. [8]

```
def standarizeData(train, test):
    sc = StandardScaler()
    #ajustar con datos de train
    sc.fit(train)
    #usar los mismos parametros de train en test
    train_ = sc.transform(train)
    test_ = sc.transform(test)

    return train_, test_
```

Listing 5: Código para estandarizar datos

5.1.2. Valoración del interés de las variables y selección de un subconjunto

Como se ha indicado anteriormente, se utilizará PCA para seleccionar un subconjunto de características relevantes y reducir así la dimensionalidad.

```
def featureSelection(train, test, percentage):
    pca = PCA(percentage)
    #ajustar con datos de train
    pca.fit(train)
    train_ = pca.transform(train)
    #usar los mismos parametros de train en test
    test_ = pca.transform(test)
    return train_, test_
```

Listing 6: Código PCA

5.2. Selección de parámetros e hiperparámetros

Se usará *cross validation*, que consistirá en dividr el conjunto de entrenamiento en n subconjuntos. Se tomarán n-1 conjuntospara el entrenamiento y el conjunto restante para *test*. Se repetirá este proceso hasta que todos los conjuntos se hayan usado para *test* (5.4).

La validación cruzada nos servirá para seleccionar los parámetros que nos den un porcentaje de aciertos más alto.

Se distinguen dos hiperparámetros [9]:

- `C:` inversa de la fuerza de regularización. A través de este parámetro se aplica la regularización. Es necesario regularizar para evitar sobreajuste. La biblioteca le da a C un valor por defecto, pero para mejorar la actuación del modelo buscaremos el valor de C que nos proporcione una mayor tasa de acierto (5.2). Se empleará regularización L2 o *Ridge*.

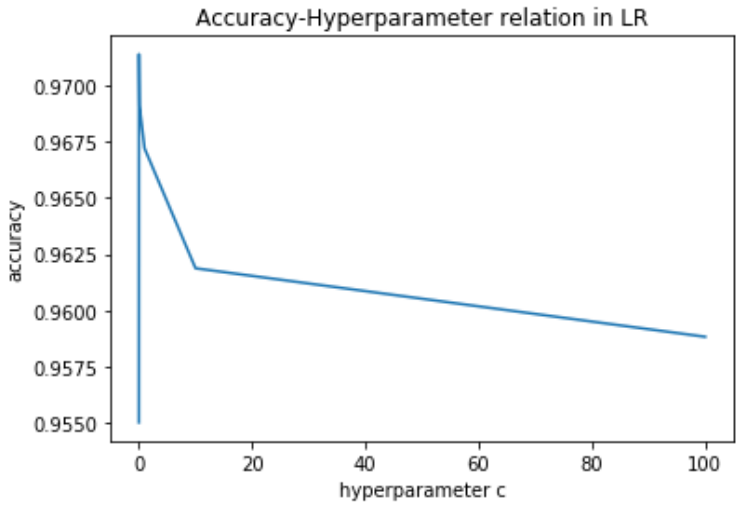

Figura 5.2: Evolución tasa de aciertos con el hiperparámetro C.

Es una muestra de que hay que ajustar muy bien el hiperparámetro C para que no se produzca sobreajuste.

- `num_splits:` número de divisiones del conjunto en la validación cruzada.

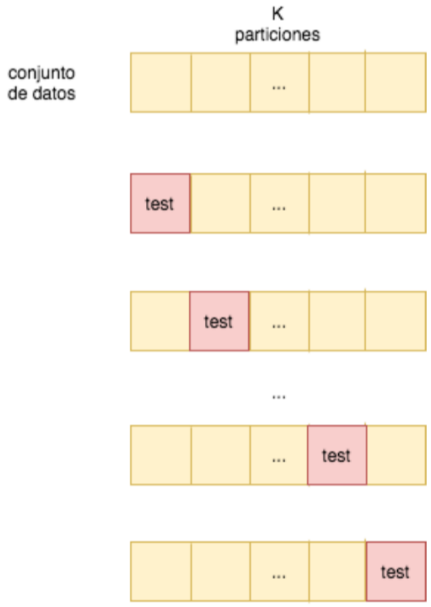

Figura 5.3: Esquema de actuación de validación cruzada

A continuación puede verse el código que se utilizará para validación cruzada (7).

```
def crossValidationLR(x, y, numsplits, hyperparameters):
```

```
    lr = LogisticRegression(solver='lbfgs', random_state=0)
    hyp = {'C': hyperparameters}
    gr = GridSearchCV(estimator=lr,param_grid=hyp, cv=numsplits)
    gr.fit(x,y)
    bestc = gr.best_params_.get('C')
    bestmean = gr.best_score_
    listofmeans = gr.cv_results_['mean_test_score']

    return bestc, bestmean, listofmeans
```

Listing 7: Código validación cruzada

5.3. Selección del modelo final

El modelo final seleccionado será regresión logística con el valor de C que se haya seleccionado a partir de la validación cruzada. En este caso, el valor es $C = 0{,}01$.

```
Starting logistic regression classification.

Cross validation finished.
Time used:  5.57351279258728
Best c: 0.01
Ein: 0.02859
```

Figura 5.4: Resultados de ejecutar validación cruzada

5.4. Error de generalización

El error de generalización es aquel que obtenemos una vez que probamos los datos de test en el modelo ajustado. Podemos ver los resultados en la figura (5.5).

```
Accuracy: 97.10366 %
Eout: 0.02896
```

Figura 5.5: Porcentaje de aciertos

El error de generalización es `0.02896`, que es un valor muy bueno. Tenemos un porcentaje de fallos muy bajo.

5.5. Matriz de confusión

Considero que la matriz de confusión es una buena métrica en el caso de la clasificación ya que nos permite ver con claridad dónde ha fallado el modelo. Como podemos ver que hay pocos falsos negativos y que principalmente los errores de clasificación se concentran el los falsos positivos (5.6).

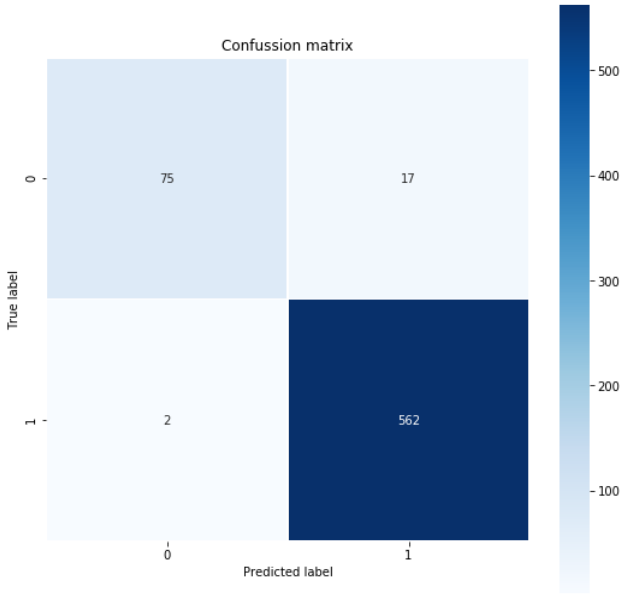

Figura 5.6: Matriz de confusión

6 Random Forest

El siguiente modelo que consideraremos será Random Forest.

Se trata de un modelo no lineal y de un predictor muy eficiente. Al igual que en el modelo anterior, se estandarizarán los datos y se hará una serie de características con PCA. Este proceso no se volverá a explicar.

No se aplicará regularización, ya que no hay ninguna variable en la función que lo regule.

6.1. Selección de parámetros e hiperparámetros

Se tendrán en cuenta varios parámetros [10]:

- Número máximo de características a considerar en la división (`max_features`). Se escogerá `sqrt`.
- Número de árboles (`n_estimators`): es el hiperparámetro cuyo valor determinaremos en la validación cruzada.
- Boostrap: `True` para aplicar bootstrapping.

Al igual que en regresión logística, se aplicará validación cruzada para estimar el hiperparámetro `n_estimators`. El número de árboles que se considerará serán las decenas entre 10 y 100.

A continuación puede verse el código que se utilizará para validación cruzada (8).

```
def crossValidationRF(x, y, numsplits, hyperparameters):
    rf = RandomForestClassifier(bootstrap=True, max_features='sqrt'
        , random_state=0)
    hyp = {'n_estimators': hyperparameters}
    gr = GridSearchCV(estimator=rf,param_grid=hyp, cv=numsplits)
```

```
    gr.fit(x,y)
    best_est = gr.best_params_.get('n_estimators')
    bestmean = gr.best_score_
    listofmeans = gr.cv_results_['mean_test_score']

    return best_est, bestmean, listofmeans
```

Listing 8: Código validación cruzada

6.2. Selección del modelo final

El modelo final será aquel que nos haya dado mejor resultado en la validación cruzada. En este caso, el número de árboles escogido es 70.

```
Time used:  221.73224115371704
Selected n_estimator:  70
Ein: 0.02478
```

Figura 6.1: Resultados de ejecutar validación cruzada.

Como puede verse, encontramos un máximo en 70 y luego la tasa de aciertos volvió a bajar. Es una muestra de que hay que ajustar muy bien los hiperparámetros para que no se produzca sobreajuste.

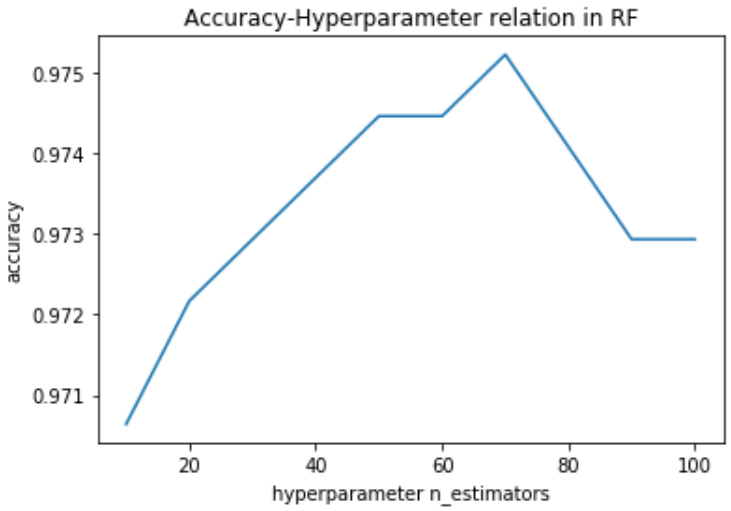

Figura 6.2: Evolución de la tasa de acierto con el número de árboles.

6.3. Error de generalización

Probaremos el modelo con el subconjunto de test para obtener un error de generalización de `0.02438`, lo que es un buen valor.

```
Accuracy: 97.56098 %
Eout: 0.02439
```

Figura 6.3: Porcentaje de aciertos

6.4. Matriz de confusión

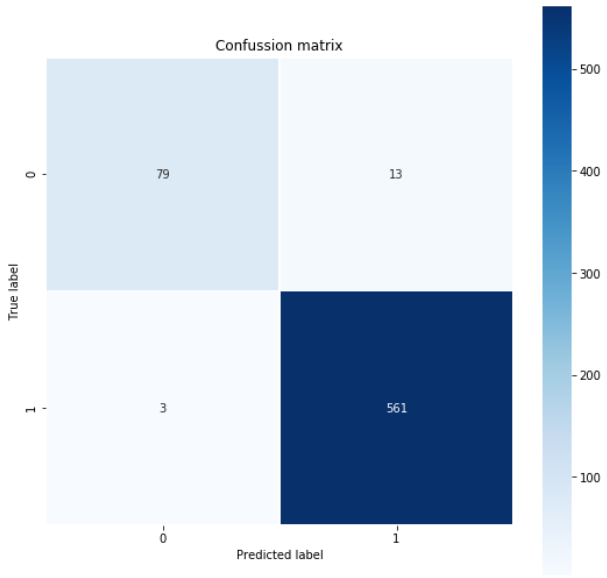

Figura 6.4: Matriz de confusión.

7 Support Vector Machine

Por último ajustaremos un modelo SVM.

El proceso utilizado será similar al usado en los modelos anteriores: estandarización los datos, reducción de la dimensionalidad y estimación de los parámetros a partir de validación cruzada. El núcleo utilizado será el Gaussiano.

A partir de uno de los artículos en los que se cita la base de datos que se utiliza se hace referencia al deterioro de SVM con dimensionalidades más bajas, ya que conduce a una reducción del número de vectores soporte [11]. Se probará si este comportamiento es en efecto verdadero.

7.1. Selección de parámetros e hiperparámetros

Se tendrán en cuenta los siguientes parámetros [12] [13]:

- `kernel`: se utilizará `rbf`, es decir, el núcleo RBF-Gaussiano.
- `C`: se aplicará la regularización a través de este parámetro, al igual que en regresión logística. Variará en el intervalo [0,1] [13]

7.2. Con reducción de dimensionalidad

Probamos en primer lugar qué ocurre aplicando PCA. Según el artículo, el resultado tiene que ser peor que sin reducir las dimensiones.

```
Cross validation finished.
Time used:  37.17579793930054
Best c: 1
Ein: 0.05757
```

Figura 7.1: Validación cruzada sin reducción de dimensionalidad.

7.3. Error de generalización

```
Using testing data...

Accuracy: 93.90244 %
Eout: 0.06098
```

Figura 7.2: Error de generalización y porcentaje de acierto.

7.4. Matriz de confusión

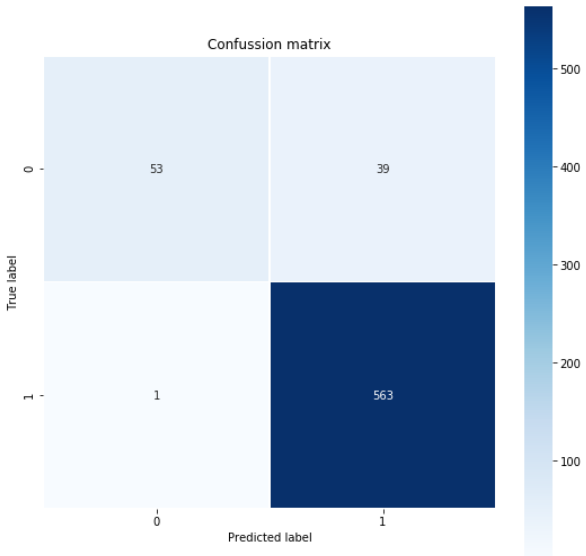

Figura 7.3: Matriz de confusión

7.5. Sin reducción de dimensionalidad

A continuación podemos ver el resultado de la ejecución sin reducción de dimensionalidad.

```
Cross validation finished.
Time used:  253.8827838897705
Best c: 1
Ein: 0.03393
```

Figura 7.4: Validación cruzada con reducción de dimensionalidad.

7.6. Error de generalización

```
Using testing data...

Accuracy: 96.03659 %
Eout: 0.03963
```

Figura 7.5: Error de generalización y porcentaje de acierto.

7.7. Matriz de confusión

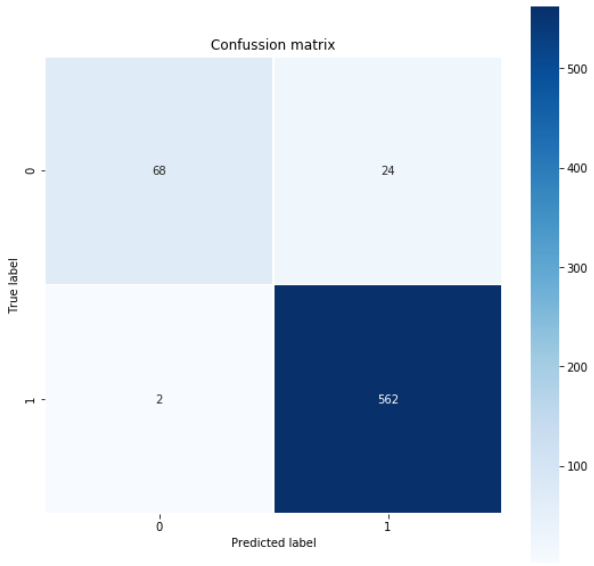

Figura 7.6: Matriz de confusión

7.8. Valoración

Se puede comprobar que en efecto, el resultado aplicando PCA es peor que si se trabaja con todos los datos. Sin embargo, el tiempo que se emplea en este algoritmo teniendo en cuenta todas las dimensiones hace que sea, en mi opinión, inviable.

8 Conclusiones

Se ha demostrado la importancia de la reducción de la dimensionalidad, la regularización y del procesamiento de datos adecuado. Dimensionalidades demasiado altas o hiperparámetros inadecuados pueden suponer sobreajuste. Por otra parte, no procesar los datos adecuadamente nos puede llevar a resultados erróneos o en el caso de PCA, a un gran sesgo.

Estos tres métodos han obtenido resultados similares. Se trata en mi opinión de buenos resultados, no creo que sea necesario una precisión más alta en un problema de estas características ya que se están tratando anuncios de internet, no enfermedades, por ejemplo.

Además, a la vista de estos resultados, considero que el mejor modelo es el de *Regresión Logística*, ya que para resultados similares es el que menos tiempo tarda en hacer predicciones. *Random Forest* y *SVM* manejan tiempos del orden de minutos, lo que considero que es innecesario teniendo un algoritmo lineal más sencillo y rápido que nos proporcione una ejecución prácticamente igual de acertada.

Referencias

[1] https://archive.ics.uci.edu/ml/datasets/Internet+Advertisements

[2] Apuntes de la asignatura.

[3] Documentación de scikit-learn.
https://scikit-learn.org/stable/index.html

[4] *Imputation of missing values*
https://scikit-learn.org/stable/modules/impute.html

[5] *e-Chapter 9. Learning Aides.* Yaser Abu-Mostafa, Malik Magdon-Ismail, Hsuan-Tien Lin. Enero 2015.

[6] *Importance of feature scaling*
https://scikit-learn.org/stable/auto_examples/preprocessing/plot_scaling_importance.html

[7] *Notas de clase de Andrew Ng*
https://cnx.org/exports/4c74010a-38d0-4a6a-9a11-4dfe8a7cbf96@4.2.pdf/machine-learning-4.2.pdf

[8] *Standarization*
https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

[9] *Logistic regression*
https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

[10] *Random Forest Classifier*
https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[11] *Experiments with Random Projections for Machine Learning.* Dmitriy Fradkin and David Madigan. 2003

[12] *Support Vector Machine*
https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC

[13] *e-Chapter 9. Support Vector Machines.* Yaser Abu-Mostafa, Malik Magdon-Ismail, Hsuan-Tien Lin. Enero 2015.