Smart_Hours.

Documentación

Integrantes del grupo: Pau Albiol, David Álvarez, Guillermo Creus.

Mesa: 4.

Fecha entrega: 30/05/2018.

Índice.

1. Objetivos.

Creación de un horario de estudio que se adapte a los objetivos académicos del cliente, respetando sus horas ocupadas y mediante el análisis de datos de estudiantes reales de ETSEIB similares a él.

2. Alcance.

Funcionalidades

Smart_Hours genera un horario y ofrece cuatro tablas y un gráfico que contienen datos de interés enfocados al cliente. Las horas del horario están distribuidas de manera adecuada para que cliente pueda cumplir con sus objetivos académicos. Para crear el horario, el cliente debe interactuar con la web rellenando tres formularios.

1. El primer formulario, consiste en indicar qué horas de la semana (de lunes a viernes, de 9:00 h a 23:00 h) tiene ocupadas el cliente para que no se asignen horas de estudio en dichas horas no disponibles.
2. En el segundo, éste debe elegir su rendimiento e introducir las asignaturas que quiere en su horario. El parámetro del rendimiento indica “el objetivo”. Hay cuatro opciones a elegir:
    - **Mínimo**: Aprobar con 5 las asignaturas elegidas.
    - **Medio**: Mantener la nota media del cliente o aprobar con 5 en caso de que esta media sea de

suspenso.
    - **Alto**: Aumentar la nota media en un 10% o aprobar con 5.5 en caso de que esta media sea de suspenso.
    - **Manual**: El cliente puede introducir qué notas quiere para cada asignatura.

3. En el tercer formulario se pide la nota de selectividad y las diez asignaturas de primero (es el Historial académico del cliente). Este parámetro permite predecir las notas que va a sacar el cliente y calcular cuán lejos está de sus objetivos.

Finalmente se generará el horario y se mostrarán las distintas tablas. Estas son:

- **Fin de semana**: Tabla que indica las horas de estudio a dedicar a cada asignatura durante el fin de semana.
- **Notas**: Tabla donde de cada asignatura elegida, se da información sobre la nota predicha, la nota deseada (es decir el objetivo) y la media global.
- **Distribución de horas de estudio**: Gráfico tipo *3d-Pie* que representa en porcentajes, el tiempo de estudio de cada asignatura (respecto al total de horas de estudio).
- **Estudiantes con notas más cercanas**: Tabla que compara las notas del Historial académico del cliente con el de los doce estudiantes más cercanos a él.
- **Notas de las asignaturas matriculadas de los estudiantes más cercanos**: Tabla que compara las notas predichas del cliente frente a las notas de los doce estudiantes más cercanos a él.

Otras funcionalidades (más técnicas)

El horario no muestra todas las horas si estas están vacías. Es decir, si el cliente estudia a partir de las 18:00 h, el horario no mostrará las horas vacías de 9:00 h a 18:00 h. Por comodidad, para rellenar los chechboxs en el apartado de selección de Horas ocupadas, es posible clicar y arrastrar con el ratón haciendo menos tedioso este formulario.

Condiciones y limitaciones

Como mínimo el cliente debe introducir una nota en el parámetro de Historial académico. Sólo se aceptan caracteres numéricos. En caso de decimales se debe indicar con un punto. (Por ejemplo: 7.5).

3. Análisis del problema y descripción de la solución.

Descripción de los problemas

1. Problema del filtrado:

Se ha partido de tres bases de datos que contenían resultados académicos de estudiantes del ETSEIB. La primera de ellas contenía las notas de selectividad, la segunda las notas de la fase inicial y la tercera las notas de la fase no inicial (hasta Q1 17/18). Debido que no siempre hay los mismos estudiantes, hay más datos de los necesarios y puede que un estudiante tenga múltiples notas de la misma asignatura (puede ser repetidor) se ha procedido a un filtrado de datos.

2. Problema de predicción:

Dado que un factor para calcular las horas de estudio es la nota predicha del cliente, obviamente se necesitan realizar predicciones (de una variable numérica continua). Para ello se ha utilizado el método de K-Vecinos más cercanos.

3. Problema de asignación de horas:

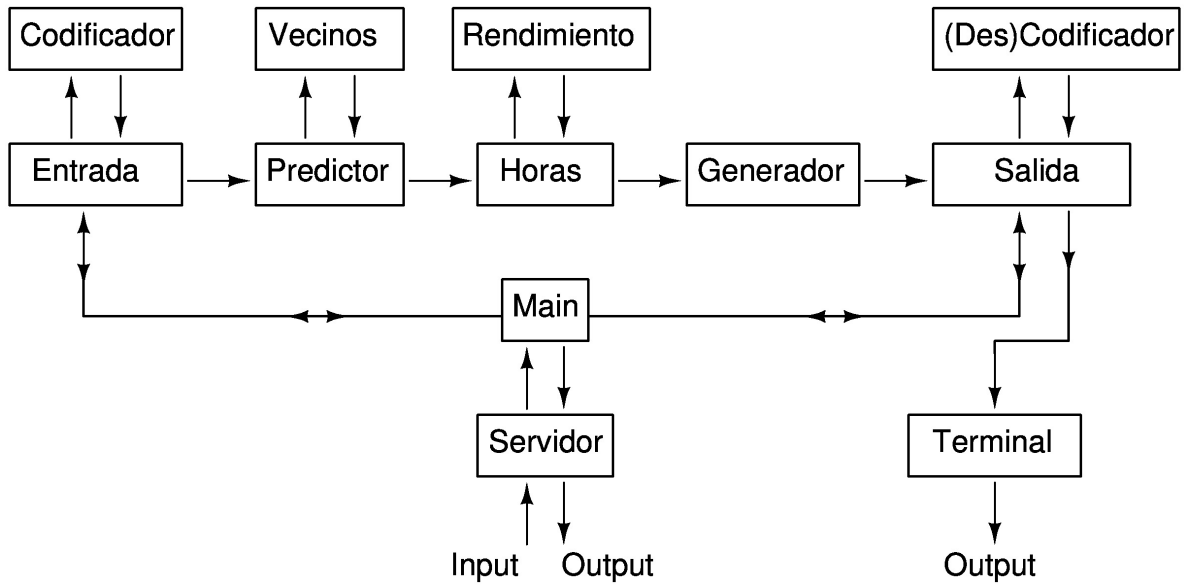
Una vez predichas las notas, se deben comparar con los objetivos del cliente y traducir a “horas de estudio”.

4. Problema de ordenación de horas:

Obtenidas las horas de estudio para cada asignatura, estas se deben distribuir en un horario de forma ordenada y lógica (evitar problemas del tipo: en un día hay que estudiar 4 horas seguidas de la misma asignatura).

Solución propuesta (Planteamiento global de funcionamiento)

El funcionamiento general del código, desde que el usuario introduce el input hasta que el output se genera en pantalla está resumido en el siguiente diagrama.



Main

Función principal que controla el programa. El funcionamiento es el siguiente:

1. Convertimos el input que el usuario entra a un input utilizable por el programa (con el script de **entrada**).
2. Predecimos las notas de las asignaturas que el usuario se ha matriculado (con el script **predictor**).
3. Obtenemos las notas deseadas de acuerdo con el rendimiento introducido por el usuario (con el script **rendimiento**).
4. Con los datos obtenidos calculamos las horas de estudio (con el script **horas**).
5. Generamos el horario de estudio (con el script **generador**).
6. Imprimimos resultados por la terminal (con el script **terminal**).
7. Generamos output utilizable por el servidor (con el script **salida**).

Entrada, Salida y Terminal

Distintas funciones para dar formato al input y al output y para imprimir el output en la terminal.

Codificador

Consiste en un codificador y en un descodificador que permiten codificar el nombre de una asignatura con su código (CODAS) y al revés. Los datos se encuentran en 'databases/asignaturas.csv'.

Vecinos

Encuentra los alumnos que están más cercanos al usuario (los datos están en 'databases/data.csv'). Para ello, buscamos a los alumnos que tienen una distancia más pequeña al usuario (entendiendo la distancia como las diferencias de las notas elevadas al cuadrado).

Predictor

Usando el script de **vecinos** hace una predicción de las notas que el usuario obtendrá en las asignaturas que se ha matriculado. Para ello (para cada asignatura) hacemos una media de las notas que han obtenido los alumnos más cercanos al usuario.
Además, para mejorar la fiabilidad, calculamos la media de la diferencia de notas (con signo) entre usuario y vecinos. De esta manera, este factor corrector lo añadimos a la predicción y podemos dar una predicción más realista (es especialmente útil para notas muy bajas o muy altas, donde nuestra base de datos es escasa).

Rendimiento

Calcula notas deseadas de acuerdo con el rendimiento introducido por el usuario:

- **Mínimo**. Aprobar con 5 todas las asignaturas.
- **Medio**. Mantener nota media del usuario o aprobar con 5 en caso de que su media sea de suspenso.
- **Alto**. Aumentar nota media del usuario en un 10% o aprobar con 5.5 en caso de que su media sea de suspenso.
- **Manual**. El usuario ya habrá introducido manualmente las notas que desea.

Horas

Dadas las notas esperadas (predictor) y las notas deseadas (rendimiento), calcula el número de horas que el usuario debería estudiar por asignatura a la semana. Para ello, usamos como base que cada crédito ECTS (*European Credit Transfer and Accumulation System*) corresponde a unas 20-25 horas de trabajo por parte del estudiante. Nosotros, descontando el número de horas de trabajo en el aula, hacemos la correspondencia 1 ECTS - 0,8 horas estudio semanales (para una persona media).
Ahora bien, comparando la nota deseada con la esperada (predicha por nuestro programa), ajustamos este número de horas a uno más realista y personalizado para el usuario.

Generador

Con las restricciones de horario introducidas por el usuario y con las horas de estudio semanales por asignatura (dadas por script horas), genera un horario de estudio. Distribuyendo las horas entre la semana y también en el fin de semana.

4. Herramientas e implementación.

Herramientas de soft usadas

Pandas

Para el filtrado de datos se ha usado pandas, una librería de Python destinada al análisis de datos que proporciona unas estructuras de datos flexibles y que permiten trabajar con ellos de forma eficiente. Más concretamente se ha trabajado con DataFrame (los datos se estructuran en forma de tablas).

Flask

Para crear la aplicación web se ha usado Flask. Flask es un framework minimalista escrito en Python que permite crear aplicaciones web con un mínimo número de líneas de código (basado en el motor de templates Jinja2).

GitHub

Para trabajar en equipo de eficiente se creó un repositorio público en GitHub (https://github.com/David-98/Horario). Github es una plataforma de desarrollo colaborativo para alojar proyectos utilizando el controlador de versiones Git.

Implementación

Comentaremos ahora cómo hemos implementado los programas más importantes. El método general usuado está resumido en el apartado 3 en la descripción de la solución.

1. Implementación del filtrado:

Se abren los tres ficheros:

```
    dsele = pd.read_csv('sele.csv')
          dini = pd.read_csv('ini.csv')
          dnoini = pd.read_csv('noini.csv')
```

En el fichero de notas de selectividad, se eliminan los datos con código de estudiante repetido (excepto la última nota):

```
    dsele = dsele.drop_duplicates(['CODEX'],keep='last')
          dsele = dsele.reset_index(drop=True)
```

Se eliminan los estudiantes que no estan en la fase inicial y fase no inicial (personas que hayan abandonado la carrera), y también se eliminan las notas de asignaturas repetidas:

```
     dini = dini[dini['CODEX'].isin(dnoini.CODEX)]
          dini = dini.drop_duplicates(subset=['CODEX','CODASS'],keep = 'first')
          dini = dini.reset_index(drop=True)
          dnoini = dnoini[dnoini['CODEX'].isin(dini.CODEX)]
          dnoini = dnoini.drop_duplicates(subset=['CODEX','CODASS'],keep = 'first')
          dnoini = dnoini.reset_index(drop=True)
```

Se procede a combinar los tres ficheros en uno mediante dos pasos:

```
    dfm = pd.merge(dini, dnoini, how = 'outer')
          dfm = pd.merge(dsele, dfm, on = 'CODEX')
          dfm = dfm.sort_values(by = ['CODEX', 'CODASS'])
```

Se ordena el archivo de la forma:

```
        CODEX   SELE    CODASS  NF
          243050        11.73   240011  6.2
          243050        11.73   240012  7.2
          243050        11.73   240013  6.0
          243050        11.73   240014  3.6
          243050        11.73   240015  8.3
          243050        11.73   240021  8.0
          ...     ...     ...     ...
          243124        11.29   240011  5.6
          243124        11.29   240012  7.0
          243124        11.29   240013  5.3
          243124        11.29   240014  5.7
          243124        11.29   240015  7.9
          243124        11.29   240021  6.0
```

Se guarda el archivo como data.csv:

```
    dfm.to_csv('data.csv', encoding='UTF-8', index=False)
```

2. Implementación de la predicción:

Está dividido en dos ficheros, el que nos encuentra los vecinos más cercanos al usuario y el que se encarga de la predicción.
El script de **vecinos** de lo que se encarga es de calcular las distancias entre el usuario y todos los alumnos de nuestra base de datos (previamente filtrada para quedarnos con los que han cursado las mismas asignaturas que el usuario). Para ellos, hay dos funciones importantes. La primera, la que nos calcula la distancia entre usuario y un alumno concreto.

```
def distance(row, notas, matrix, sele):
          distance = 0
          for j in range(0, len(notas)):
          ...
          distance += pow(notas[j] - matrix[row][j + 1], 2)
```

Si el usuario nos ha proporcioinado la nota de selectividad, la añadimos a la distancia.

```
if sele >= 0: distance += pow(sele - matrix[row][len(notas) + 1], 2)
```

Esto nos permite hallar los vecinos más cercanos al usuario.

Una vez que ya hemos encontrado los vecinos más cercanos, para cada asignatura que el usuario se ha matriculado, obtenemos la media de las notas que han obtenidos los vecinos más cercanos. Esto lo hace el siguiente programa del script **predictor** (*matrix* es donde tenemos guardada la información de los vecinos).

```
def predict_grades(nearest, codas, notas, matrix, k, diff, sele):
          ...
          grades = [0]*len(codas)
          for j in range(0, len(codas)):
          ...
          for i in range(0, k):
          grades[j] += matrix[nearest[i][1]][j + 1]
          grades[j] /= k
          grades[j] += diff
          ...
          return grades
```

En este programa la variable *diff* correspondería a una medida de si (de media) los vecinos que hemos encontrado están (en notas) por encima o por debajo del usuario. De esta manera, este factor corrector nos permite hacer una mejor predicción para intervalos de notas en los que tenemos pocos alumnos en nuestras bases de datos (especialmente útil para notas muy pequeñas o muy altas). Esta diferencia se calcula de la siguiente manera:

```
diff_tot = 0
          for i in range(0, k):
          row = nearest[i][1]
          diff = 0
          for j in range(0, len(notas)):
          ...
          diff += notas[j] - matrix[row][j + 1]
          diff_tot += diff/l_curs
          return diff_tot/k
```

Es decir, hacemos una media de las diferencias de notas entre usuario y vecinos (los k más cercanos) para todas las asignaturas.

3. Implementación de asignación de horas:

La función "hours main" convierte las notas deseadas en horas de estudio mediante una función exponencial. La función "hours main" devuelve un vector con las horas a estudiar. Como hemos explicado anteriormente (apartado 3) hacemos una aproximación inicial de 1 ECTS - 0,8 horas de estudio semanales.

```
    def hours_main(notas_des, notas_esp, notas_med, l_des, creds):
          ...
          hours = []
          for i in range(0, l_des):
          hours.append(creds[i]*0.8)
```

Evitando el rápido crecimiento de la exponencial, se multiplican las horas a estudiar por un factor que depende de la nota deseada:

```
    if -notas_des[i] - notas_esp[i] < 2:
          hours[i] *= pow(1.12, 1.7*(-notas_des[i] - notas_esp[i]))
          elif -notas_des[i] - notas_esp[i] < 4:
          hours[i] *= pow(1.12, 1.1*(-notas_des[i] - notas_esp[i]))
          elif -notas_des[i] - notas_esp[i] < 6:
          hours[i] *= pow(1.12, 0.9*(-notas_des[i] - notas_esp[i]))
          else:
          hours[i] *= pow(1.12, 0.7*(-notas_des[i] - notas_esp[i]))
```

Se aplican factores de corrección para las notas muy bajas:

```
    if -notas_des[i] <= 1:
          hours[i] = 0
          elif -notas_des[i] <= 1.5:
          hours[i] *= 0.5
          elif -notas_des[i] <= 2:
          hours[i] *= 0.75
```

Se aplica un redondeo a las horas de estudio finales y se impone un máximo de 25 horas semanales:

```
    hours[i] = int(round(hours[i], 0))
          if hours[i] >= 25:
          hours[i] = 25
```

4. Implementación de ordenación de horas:

Lo primero que hacemos es hacer una primera asignación de horas de estudio al fin de semana, más concretamente asignamos 2/7 del total de horas de estudio. Después, tratamos de distribuir en bloques de dos horas por asignatura, horas de estudio a lo largo de la semana. Para que queden distribuidas de manera homogénea, lo hacemos siguiendo la siguiente secuencia: Lunes, Miércoles, Viernes, Martes, Jueves.

```
sec = [0, 2, 4, 1, 3]
          j = 0  # Iterador para la secuencia de días.
          for asig in range(0, n):
          exit = 0
          while hours[asig] > 1 and exit < 5:
          if j > 4: j = 0
          if put2(asig, sec[j], horario):
          hours[asig] -= 2
          hras_est_dia[sec[j]] += 2
          else: exit += 1
          j += 1
```

Por último, tras evitar asignaturas con demasiadas horas en el fin de semana, procedemos a distribuir ahora de 1 hora en 1 hora.

```
for asig in range(0, n):
          finished = False
          while hours[asig] > 0 and not finished:
          day = next_minimum_uncomplete(horario, hras_est_dia)
          if day == -1: finished = True
          elif put1(asig, day, horario):
          hours[asig] -= 1
          hras_est_dia[day] += 1
```

Por último, guardamos las horas de estudio que no han entrado en la semana como horas de fin de semana.

```
for i in range(0, n):
          finde[i] += hours[i]
```

5. Planificación y costes.

Para realizar el proyecto de **Smart_Hours**, una de las primeras cosas que se creó fue un plan de ruta cuya planificación abarcaba cerca de dos meses de desarrollo. Las horas dedicadas en **Smart_Hours** han sido para cada integrante:

- 5 horas de clase de teoría dónde se aprendió sobre el uso de pandas y flask.
- Un promedio de 55 horas de autoaprendizaje.

Es decir, en total unas 180 horas.

El único coste del proyecto ha sido el consumo energético para alimentar los ordenadores. Suponiendo que un ordenador usa 100W por 1 hora diaria a 0,12 € el kWh, calculando para un integrante el coste mensual es de 0,43 € al mes. Dado que el proyecto ha durado 2 meses y **Smart_Hours** ha sido desarrollado por 3 integrantes, el coste energético estimado es de 2,6 €.

6. Resultados y conclusiones.

Para realizar este proyecto, una buena comunicación y coordinación entre el equipo ha sido imprescindible. La estrategia de haber dividido y planificado el proyecto en pequeños problemas y asignar roles a cada integrante ha resultado muy eficiente. Ha simplificado las cosas pues cada integrante se focalizaba en un problema y en caso de no saber continuar, entre todos se ha buscado una solución o propuesto alternativas para solucionar (o en algún caso evitar) el problema.

Un gran impacto de este proyecto ha sido el de autoaprendizaje pues siempre que aparecía algún inconveniente, en internet se hallaba la solución.

También se han adquirido competencias genéricas sobre la comunicación oral y escrita. Ha habido una clara evolución en cómo se han realizado las presentaciones orales y el estilo de los PDF's que se proyectaban.

7. Bibliografía.

https://stackoverflow.com/

https://www.w3schools.com/

https://pandas.pydata.org/pandas-docs/stable/10min.html