MAL O2 Journal

ITMAL-01 Forår 2022

Journal over opgaver fra L03-L06

Afleveret af: Gruppe 21

Retning	Navn		Studie ID	AU-nummer
E	Lukas Kezic	LKE	201906917	AU637735
E	Rasmus Holm Lund	RHL	201900058	AU630898
E	Jakob Peter Aarestrup	JPA	201907898	AU632998
E	Frederik Thomsen	FT	201906146	AU637451

Kontaktperson:

Retning	Navn	Studie ID	Email
E	Rasmus Holm Lund	201900058	201900058@post.au.dk

Aarhus Universitet
23. marts 2022

Indholdsfortegnelse

1 L03 - Supergruppe diskussion

1.1 Look at the Big Picture

Når man starter ud med et machine learning project man får fra sin leder er det vigtigt at se på det større billede. Heraf hvilken måde man skal gribe problemet an. I bogen bliver der brugt eksemplet omkring at finde medianen for huspriser i Californien USA. Metrics som vil blive brugt til modellen er population, median inkomst og huspriser. Dette bliver gjort for hver block gruppe. En block gruppe er på 600 til 3000 mennesker. Nu ved man hvad for noget data der skal bruges, men hvad er målet med denne model som du skal lave? Denne model skal bruges til at gætte på median prisen for huse i Californien og gives til et andet machine learning system, som vil finde ud af om det er værd at investere i de forskellige block områder. Hvorfor lave en model overhovedet? Jo, ligenu bliver det gjort manuelt af nogle eksperter, hvor de selv sidder og estimere, men de estimere forkert med omkring 20%. Derfor er det nu din opgave at lave en model som kan gøre det bedre. Der er mange måde at gribe det an på, men du vælger regression task, da dit output er med i dit dataset. Derudover er det kun en værdi som skal findes. Derfor er det et univariated regressions problem, istedet for f.eks. en multivariated regression. Hertil skal der vælges performance measures. Her kunne RMSE og MAE bruges. RMSE bruges til at finde ud af hvor mange fejl, der typisk bliver lavet i ens estimering. MAE bliver brugt når der er meget afvigende data, det kunne være districter eller blokke som ligger langt udenfor normen af datasættet. Til sidst er det vigtigt at være helt sikker på at den data du får ud er den rigtige. Downstream modellen kunne f.eks. opdele det i kategorier og derved ville median værdien for de forskellige distrikter ikke kunne bruges. Derfor er det vigtigt at have hele overblikket for ikke at spilde mange måneder på den forkerte løsning.

1.2 Get the Data

For at kunne starte på koden skal der laves et workspace. Hertil vil der blive brugt python. Ydermere installeres forskellige moduler vha. pip install. Heraf f.eks. jupyter. Nu hvor dette er sat op kan man importere dataen. Denne data kan f.eks. hentes ved at lave en fetch funktion i python. Med denne data vil der nu kunne blive lavet vores model fra det tidligere kapitel. Det kan også ses at der er noget afvigende data, idet at der er 20640 instances, hvor 207 af distrikterne ikke har soveværelser, som vil blive kigget på senere. For at kigge på dataen kan funktionen describe() bruges. Man kan også plotte dataen i et histogram. Dette er for at få et overblik over sin data. Når man vælger sin testdata er det vigtigt er gøre det tilfældigt. Det kunne være 20% af et tilfældigt dataset, da det vigtigt man ikke er bias. Funktioner som kan bruges til random data er f.eks. np.random.seed() og np.random.permutation. For at være sikker på at din data opfylder 20% standarden kan der også tilføjes en hash. Nu er den random data generet nu skal det opdeles i subset. Her kunne Scikit-Learn bruges, hvor de to funktioner train_test_split() og split_train_test er gode at bruge. Derudover er det vigtigt at opdele din data i stratums så det er læseligt data. I vores tilfælde er det median priserne mellem 15000 dollars til 60000 dollars, der skal opdeles. pd.cut kan bruges her til opdele denne data og scikitlearn's StratifiedshuffleSplit til at inddele i kategorier. Der er mange overvejelse at tage når man skal bearbejde sine data. Det er derfor også vigtigt at sætte tid meget tid af til det, for ikke at skulle rette op på fejl senere i sin model.

1.3 Discover and Visualize the Data to Gain Insights

For at få en bedre følelse for ens data, kan det hjælpe en del at visualisere dataen. Dette kan gøres på mange forskellige måder, men en universal god måde er at lave et såkaldt heatmap. Det skal dog siges at hvilket plot man bruger, er meget afhængig af ens data. I dette kapitel bliver dataen først plottet i et scatter-plot, som virker meget uoverskuelig. Ved at tilføje en ny alpha værdi, som ændre på gennemsigtigheden af punkterne i plottet, bliver det mere overskueligt, men det kan stadig blive bedre. Ved at tilføje en masse nye parameter til plot funktionen, kan vi få det der hedder et heatmap. Dette er det mest informative plot af de viste plots

indtil videre. Dette er dog kun brugbart for folk der ikke kender dataen, så de hurtigt kan forstå dataen. Derfor er man ofte mere interesseret i at bruge plots til at lave sammenligninger. Her kommer correlations ind i billedet. Vi starter med at lave et matrix af vores correlations. Dette giver os et matrix af både integers og string værdier, altså både navne og værdier. Denne matrix kan plottes, så vi plotter hver unikke position for sig selv, resultatet af dette er en matrix af plots. Man skal altså forstille sig at man nu har lavet alle tænkelige plots man kan lave ud fra ens data og sat disse ind i et plot.

1.4 Prepare the Data for Machine Learning Algorithms

Nu kan vi forberede vores Data til Machine Learning Algorithmer. Vi starter med at identificere vores muligheder, her er det en god ide at kigge på deres output for at se hvad de indeholder. I eksemplet i bogen indeholder valgmulighed 1 ingenting og 3 indeholder alt. Man skal bruge medians, dette betyder at vores valgmuligheder ikke må indeholde bogstaver. Vi får altså et simpelt array af medianer, hvis man har udregnet disse ved brug af math funktioner, er det en god ide at bekræfte dem manuelt. Dette array er altså vores training set. Dette training set kan vi transformere, så det ikke er median værdier mere. Vi kan altså genskabe vores muligheder fra starten, ved hjælp af vores transformeret training set. Man kan tilføje tekst til ens attributes ved hjælp af en encoder. Ligeledes kan man tilføje sine egne attributes, ved at lave en costum transformer. Til sidst bruger vi en pipeline til at samle alle stepsene i en.

1.5 Select and Train a Model

Når man har opdelt sit data i train og test sæt er man nu klar til at udvælge og træne en model ud fra test sættet. Der findes flere forskellige modeller, og de bør vælges ud fra hvilken type data man arbejder med. Hvis ens data f.eks. er lineært fordelt, vil det være en god idé at fit sin data ud fra en lineær regressions model. For at test sin models præstation, laver man typisk en predict så ens model forudsiger nogle værdier. Disse værdier sammenligner man med sit test-data, hvor man til sidst udregner fejlen f.eks. Mean Sqaured Error mellem de to. Man vil typisk få en meget stor fejl, hvis den model man har valgt ikke er god til at repræsentere data'en med. Dog kan man nogle gange godt have valgt en god model, men pga. man ikke har nok testdata vil ens data være overfit.

1.6 Fine-Tune Your Model

For at få et bedre resultat fra sin model kan man fitte den ved at ændre på hyperparametre. Parametrene kan man vælge manuelt, men man kan også bruge forskellige værktøjer såsom cross validation til at udregne de bedste værdier. Generelt gælder der for cross validation, at desto flere træninger på ens data man udfører, desto bedre vil ens model blive. Dog vil flere træninger medføre mere computeringstid. Cross validation virker fint til mindre datasæt, men ved større datasæt er Randomized search værktøjet bedre, da det laver flere iterationer per hyperparameter. Derudover kan man selv indstille antallet af iterationer, og dermed styre hvor meget man vil belaste sin CPU. Man kan altså lave tuning af parametrene. Man skal dog være opmsærksom på, at meget tuning af hyperparametre kan give et værre resultat end cross validtion på ukendte datasæt, da man har tunet, så de passer til et specifikt datasæt. Testen af fintuning testes på samme måde som ved test af en model. Man bedømmer altså fintuningen præstation ud fra, hvor tæt på den fittede model er på det datasæt der måles på.

1.7 Launch, Monitor, and Maintain Your System

Nu er systemet klar til launch og de sidste ting skal finpudses inden modellen kan indsættes i ens produktions miljø. Heri kan den bruges til at lave forudsigelser om f.eks. priser vha. predict() funktionen. Dette kan også gøres ved at bruge en Web service eller, der simplificerer skalering og opdatering. En tredje måde er at uploade systemet til cloud udbydere som google, der sørger for load balancering og skalering for en. Derudover skal der skrives kode, der kan monitorere systemets live præstation, der kan alarmere hvis den forringes. Dette kan være kompliceret at gøre og kan gribes an på forskellige måder. f.eks kan det ske igennem spørgeskemaer, eksperter der kigger data igennem osv. Det kan også være at kigge sit input data igennem er det som løser fejlen. Pointen er at det er vigtigt at løbende tjekke op på sit system, lave backups af modellerne, og have en effektiv fejlsøgning og monitorering.

1.8 Try It Out!

Størstedelen af arbejdet med machine learning ligger i forberedelsen. Heri skal der laves monitorering værktøjer, menneskelig evaluering, pipelines og automatisering af regelmæssig model træning. En effektiv og nøjagtig algoritme er vigtigt, men i det introducerende arbejde med machine learning er det vigtigere at have styr på disse steps inden og kombinere det med viden omkring 3-4 algoritmer. Derefter kan man tage fat i de avancerede algoritmer bagefter, når disse ting på plads.
Med disse kapitler under armen skal der nu bare findes et passende datasæt, der kan bruges til at køre hele processen igennem med. Et godt sted at starte er at kigge efter datasæts på Kaggle.com.

2 L04 - Dataanalyse

2.1 Qa - Beskrivelse af datasæt til O4 projekt

Til O4 projektet har vi valgt at arbejde med vine og deres tilhørende kvalitet. Formålet er at kunne differencere mellem gode og dårlige vine ved at kigge på de kemiske egenskaber. Dette er gældende for både røde og hvide vine, hvor hver vin har en score fra 1-10. Sammenhængen mellem scoren og de kemiske egenskaber skal skabe grundlaget for hvordan algoritmen bestemmer hvilke vine, der er gode og hvilke der er dårlige.

Til projektet er der valgt et datasæt baseret på en undersøgelse fra Portugal [Cortez et al., 2009], der beskriver typen "Vinho Verde"i både rød og hvid variant. Vinene i sættet har en række parametre, der består af deres kemiske kvaliteter som; tæthed, pH-værdi, sulfitter, alkohol og en score, der rangerer vinene mellem 1 og 10. Datasættet er fundet igennem UCI Machine Learning Repository, der er en database med datesæt, som specifikt er gode til machine learning. Selve sættet kan findes på følgende hjemmeside:

https://archive.ics.uci.edu/ml/datasets/Wine+Quality

Datasættet, der består af 4898 samples indeholder 12 parametre, hvor de første 11 er de kemiske egenskaber og den sidste er scoren. Parametrene kan ses på tabel 1:

1	Fixed acidity	7	Total sulfur dioxide
2	Volatile acidity	8	Density
3	Citric acid	9	pH
4	Residual sugar	10	Sulphates
5	Chlorides	11	Alcohol
6	Free sulfur dioxide	12	Quality (1-10)

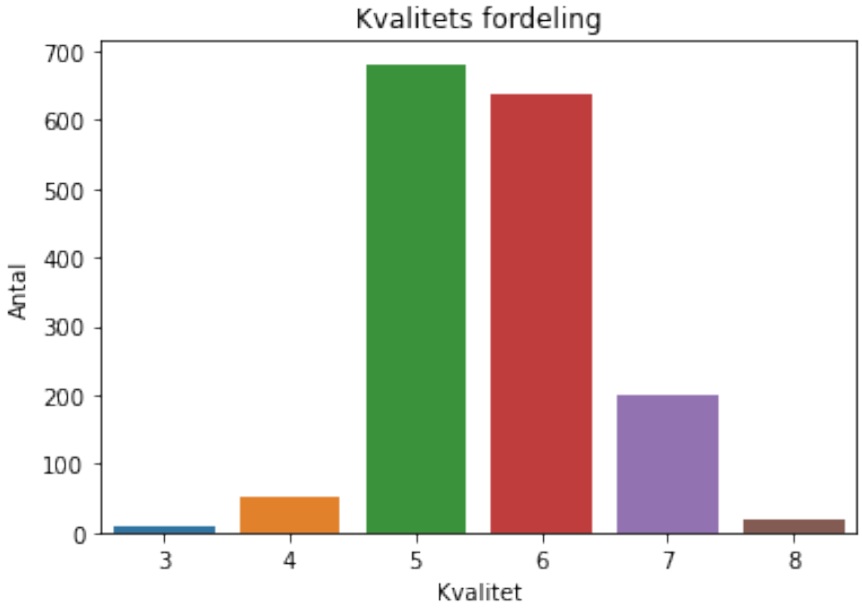
Tabel 1: Parametre i datasæt

Som udgangspunkt er der ikke en særlig stor samplesize i dette sæt, hvilket typisk godt kan skabe problemer, hvis man vil lave en succesfuld machine learning algoritme. Det kunne f.eks. være i nogle billedgenkendelses opgaver, hvor en stor samplesize kan være ret vigtig. I tilfældet med dette datasæt burde det dog ikke skabe store problemer. Derudover gælder sættet kun for en enkelt type vin, hvilket gør det svært at bruge resultaterne ift. andre typer. Samtidigt er det ikke sikkert at alle parametre er relevante, men dette giver også mulighed for at teste en feature selection method.

Datasættet kan både bruges til at lave en regression eller klassifikation, men muligheder som neural network giver ikke meget mening med dette datasæt. Som udgangspunkt er størstedelen af vinene rangeret enten som 5 eller 6 i score. Derfor vil sættet ikke være særlig effektivt til at rangere middelmådige vine ift. hinanden. Derimod burde sættet fungere en del bedre til at bestemme outliers. Dvs. burde sættet være godt til at finde de meget dårlige vine og dem som er helt exceptionelle. Det er derfor vores fokus i dette projekt.

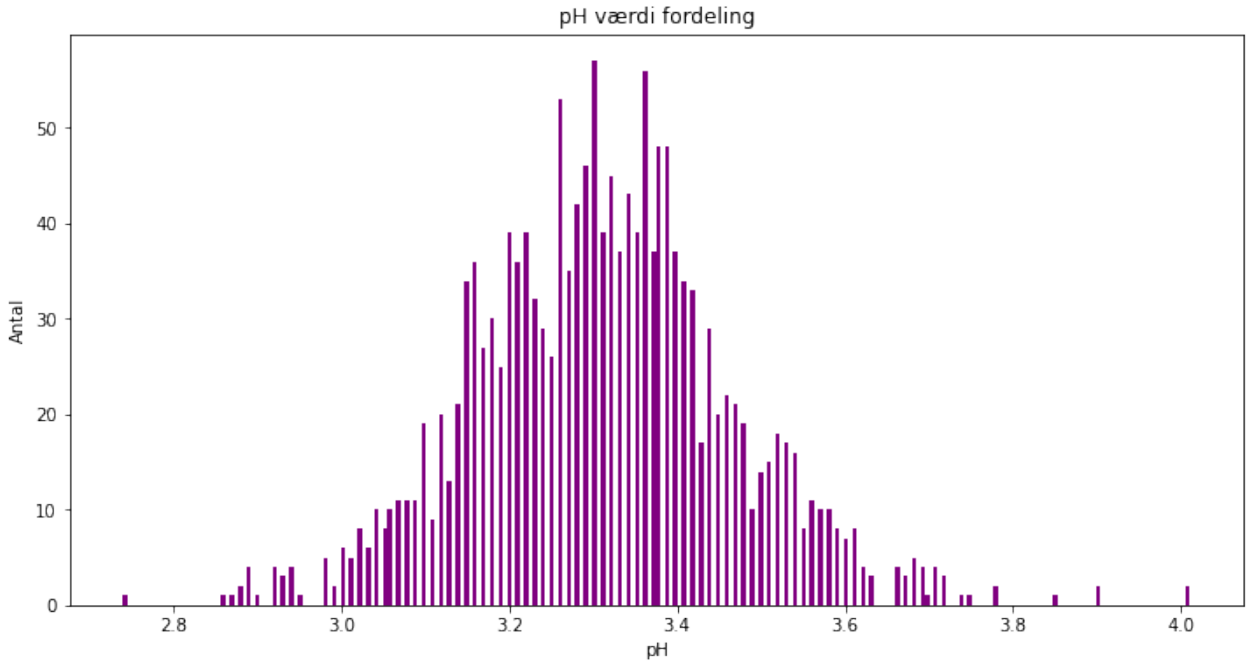
2.2 Qb - Dataanalyse af eget datasæt

På Figur 1 kan fordelingen af kvaliteten ses. Heraf at de fleste vine i vores datasæt har fået en score på 5 og 6. Derudover er der en del som har fået 7 også. Outlierne i denne fordeling er helt klart vine med scoren 3 og 8. Hvor der er flere med en score på 8 end 3.



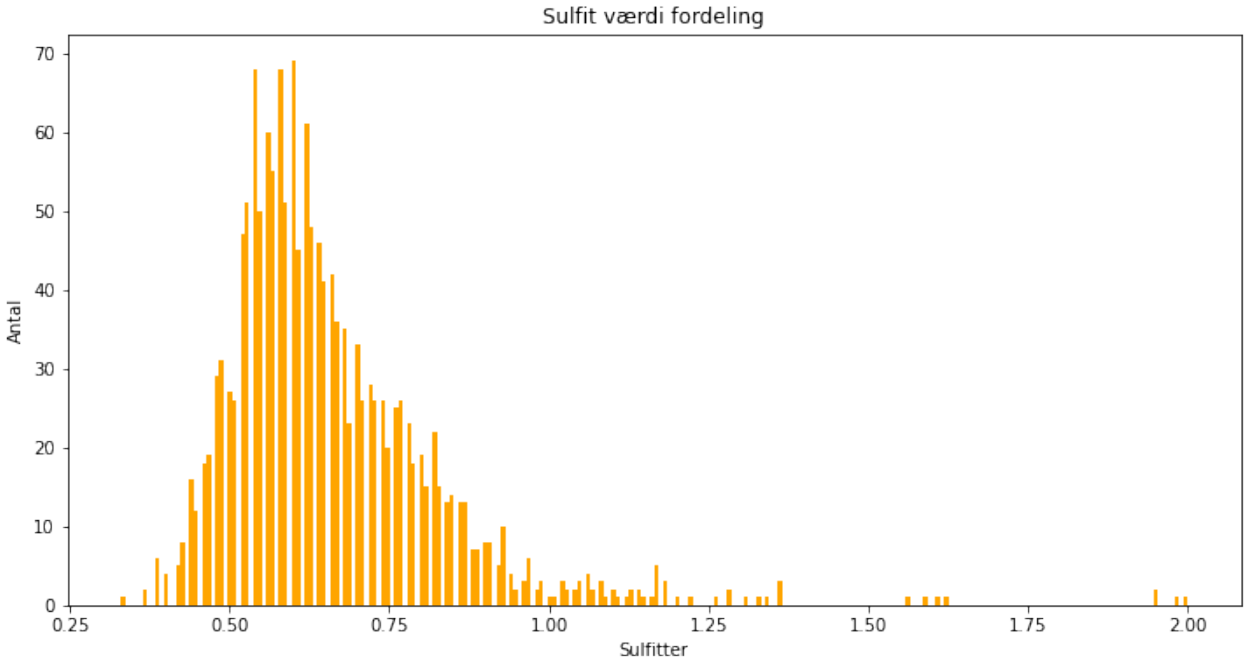
Figur 1: Kvalitets histogram

På Figur 2 er pH-værdi fordelingen vist. I denne fordeling kan ses at størstedelen af vinene har en pH-værdi mellem 3.2-3.4. Hvor at der er nogle enkelte over 4 og enkelte under 2.7. Dvs. hvis man kigger på pH-skalaen på er det meste af vinen omkring syrlig i pH-værdi fordelingen. Dog med outliers på 4 som er mindre syrlige som bevæger sig mod basis og 2.7 der er bevæger sig mod mere ætsende.



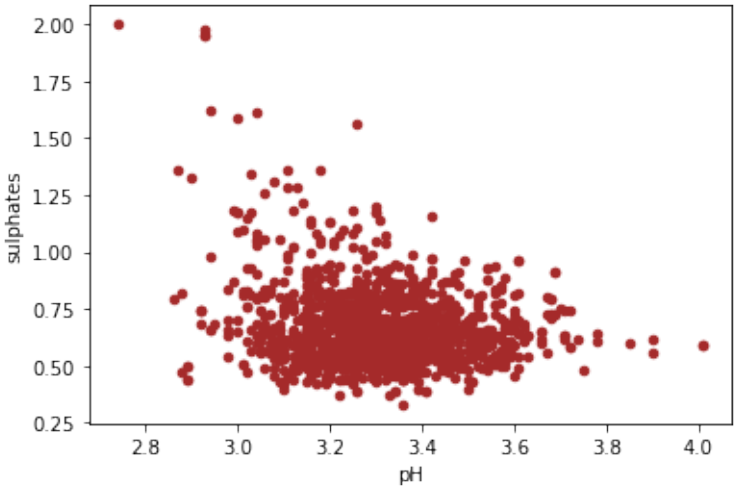
Figur 2: pH histogram

På Figur 3 kan fordelingen af sulfitter i vinene ses. Her ses den samme normalfordeling som ved pH-værdien ikke helt. Størstedelen af vinene har et lavere indhold af sulfitter, men der ses også outliers som næsten har firedobbelt så mange som medianen. Typisk for syrlige vine behøves der ikke samme mængde af sulfitter for at vinen er holdbar. Det hænger godt sammen med at størstedelen af vinene i dette sæt er i den syrlige ende.



Figur 3: Sulfit histogram

Kigger vi på Figur 4 er sammenhængen mellem pH-værdi og sulfitter også tydelig. Her har størstedelen af vinene en pH-værdi mellem 3,0 og 3,6, men samtidigt har de fleste også en sulfit værdi mellem 0,5 og 0,8.



Figur 4: pH over sulfit scatter plot

Ligesom at dataen kan plottes kan der også findes median, middelværdi og standard afvigelse for de respektive parametre. På Kode snippet 1 kan udregningen af dette ses samt outputtet. Her er det tydeligt at middelværdien er tættest på medianen for pH-værdierne og længst fra for kvalitet. Samtidigt er standard afvigelsen betydeligt lavere for pH og sulfitter.

```
1  from statistics import mean, median, stdev
2
3  quality = wine['quality']
4  sulphates = wine['sulphates']
5  pH = wine['pH']
6
7  print("Quality:")
8  print("Mean:", round(mean(quality),3))
9  print("Median:",round(median(quality),3 ))
10 print("Std deviation:",round(stdev(quality),3))
11 print("Sulphates:")
12 print("Mean:",round(mean(sulphates),3))
13 print("Median:",round(median(sulphates),3))
14 print("Std deviation:",round(stdev(sulphates),4))
15 print("pH:")
16 print("Mean:",round(mean(pH),3))
17 print("Median:",round(median(pH),3))
18 print("Std deviation:",round(stdev(pH),3))
19
20 > Quality:
21   Mean: 5.636
22   Median: 6
23   Std deviation: 0.808
24   Sulphates:
25   Mean: 0.658
26   Median: 0.62
27   Std deviation: 0.1695
28   pH:
29   Mean: 3.311
30   Median: 3.31
31   Std deviation: 0.154
```

Listing 1: Median, middelværdi og standard afvigelse

3 L04 - Pipelines

3.1 Qa - Lav en Min/max scaler til MLP

I denne opgave skal MPL () benyttes som en model for noget data om neurale netværk. For at sørge for at modellen giver en god repræsentation af dataen, skal alle værdier skaleres ned til værdier mellem 0 og 1.

For at skalere værdierne findes først forskellen mellem maksimum- og minimumsværdierne. Herefter kan forskellen beregnes, hvor de skalerede X-værdier til sidst kan beregnes. Dette er vist på Listing 2.

```
1 X_min = np.min(X)
2 X_max = np.max(X)
3 s = X_max-X_min
4
5 X_scaled = (X-X_min)/s
```

Listing 2: Skalering af X-værdier

Der laves nu en fit samt predict på den nye skalerede data, og til sidst udregnes der en score. Dette er vist på Listing 3.

```
1 mlp.fit(X_scaled ,y.ravel())
2 y_pred_mlp = mlp.predict(X_scaled)
3
4 print(f"mpl.score={mlp.score(X_scaled, y.ravel()):0.2f}")
5
6 >mpl.score=0.72
```

Listing 3: fit og predict af skaleret data samt udregng af score

Som man kan se, så fås en score på 0.72. Dette er en markant bedre score sammenlignet med scoren fra den ikke-skalerede data som blev til -7.57.

3.2 Qb - Scikit-learn Pipelines

I denne opgave skal data'en igen skaleres, men denne gang med brug af funktionen preprocessing.MinMaxScaler fra biblioteket sklearn.

Først initialiseres scaleren, hvorefter den fittes til X-værdierne. Herefter bruges funktionen transform() til at skalere M- og X-værdierne. Til sidst laves en fit og predict på den nye skalerede data, hvorefter en score udregnes. Dette er vist på Listing 4.

```
1  from sklearn.preprocessing import MinMaxScaler
2
3  scaler = MinMaxScaler()
4  scaler.fit(X)
5
6  X_scaled = scaler.transform(X)
7  M_scaled = scaler.transform(M)
8
9  mlp.fit(X_scaled, y)
10 y_pred_mlp = mlp.predict(M_scaled)
11
12 print(f"mlp.score={mlp.score(X_scaled, y):0.2f}")
13
14 >mpl.score=0.72
```

Listing 4: Skalering med MinMaxScaler

Som man kan på Listing 4, så fås en score på 0.72, der stemmer overens med scoren fra den manuelle løsning vist på Listing 3.

Der laves nu en pipeline, der indeholder 'scaler' funktionen samt 'mlp' funktionen. Først fittes det data, der ønskes at blive skaleret, som i dette tilfælde er X og y. Herefter defineres det, at mlp-funktionen ønsker at blive brugt. Dermed kan scoren af den skalerede data blive udregnet. Dette er vist på Listing 5.

```
1  from sklearn.pipeline import Pipeline
2
3  pipe = Pipeline(
4      [
5          ('scaler',MinMaxScaler()),
6          ('mlp',mlp)
7      ]
8  )
9
10 pipe.fit(X,y)
11 pipe = pipe["mlp"]
12
13 print(f"pipe.score={pipe.score(X_scaled, y):0.2f}")
14
15 >pipe.score=0.73
```

Listing 5: Skalering med pipeline

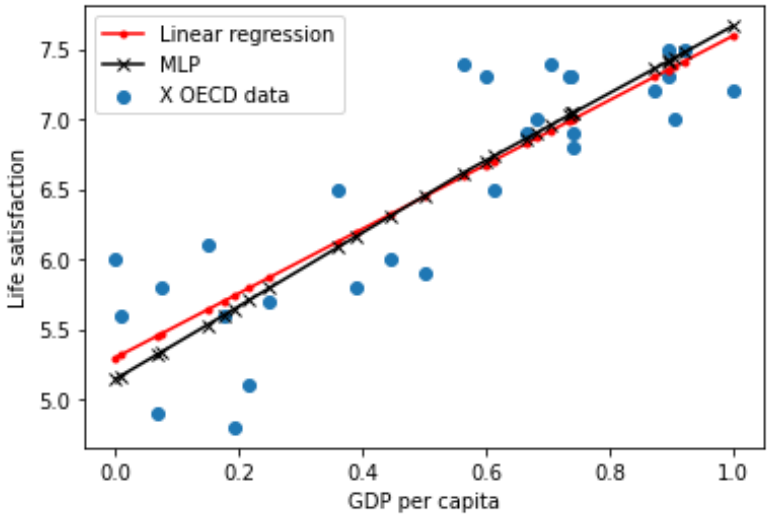
Som man kan på Listing 5, så fås en score på 0.73, der stemmer overens med de andre løsninger.

Til sidst bruges en lineær regressions model på dataen, som mlp-modellen vil blive sammenlignet med. Dette er vist på Listing 6. Derudover plottes dataen samt modellerne med den indbyggede funktion PlotModels() som vist på Figur 5, som også udprinter modellernes score.

```
1 linreg2 = LinearRegression()
2 linreg2.fit(X_scaled,y.ravel())
3
4 PlotModels(linreg2, pipe, X_scaled, y, "Linear regression", "MLP")
5
6 >Linear regression.score(X,y)=0.73
7 >MLP.score(X, y)=0.73
```

Listing 6: Lineær regression samt plot

Som man kan se, så på Figur 5, så minder den lineære regression model og mlp-modellen meget hinanden, og ser man på outputtet i Listing 6, så kan man se, at de har den samme score.



Figur 5: Lineær regressions model vs. mlp-model

3.3 Qc - Outliers samt Min-max Scaleren vs. Standard Scaleren

Min-max scaling kaldes også for normalisering og fungerer på den måde, at vi tager nogle værdier og shifter og rescaler dem, så de ligger mellem 0 og 1. Dette er godt, hvis man f.eks. arbejder med neurale netværk, hvor værdierne typisk skal være imellem 0 og 1. Dog er en af problemerne med denne metode, er at outliers kan betyde, at de andre rigtige værdier bliver sat så langt ned, at de mister deres betydning, mens outlieren får stor betydning. Dette problem oplever vi ikke med standard scaleren også kaldes standardisering. Med denne metode bliver værdierne ikke skaleret indenfor et bestemt interval, og derfor vil outliers have mindre betydning [Aurélien Géron, 2009]. Nedenfor på Listing 7 er standard scaleren blevet testet for at se om den performer bedre.

```
1  from sklearn.preprocessing import StandardScaler
2
3  scaler = StandardScaler()
4  scaler.fit(X)
5
6  X_scaled = scaler.transform(X)
7  M_scaled = scaler.transform(M)
8
9  mlp.fit(X_scaled, y)
10 y_pred_mlp = mlp.predict(M_scaled)
11 print(f"mlp.score={mlp.score(X_scaled, y):0.2f}")
12
13 >mlp.score=0.77
```

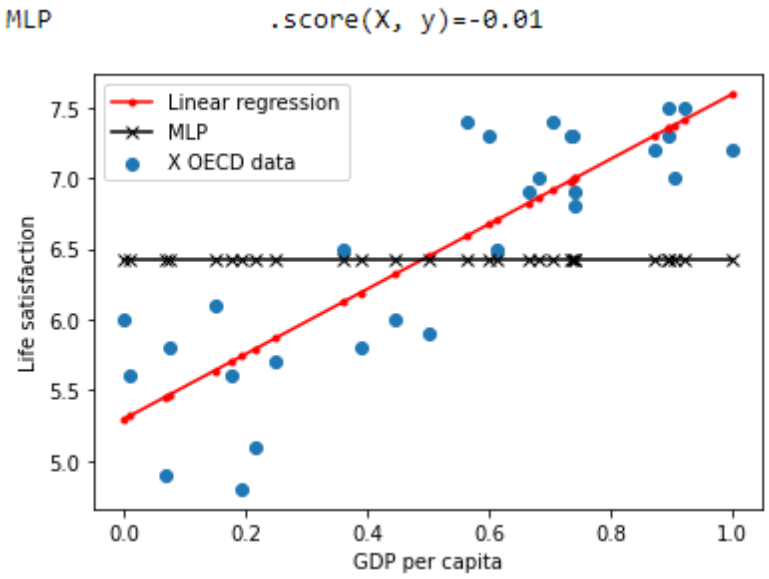
Listing 7: Test af standard scaler

Som man kan se, så får vi en score på 0.77, som er en bedre score end de 0.73 som vi får, når vi bruger Min-max scaleren. Dette skyldes nok, at datasættet indeholder en eller flere outliers, der påvirker outputtet meget, og derfor er standard scaleren bedre i dette tilfælde.

3.4 Qd - Modificér MLP'ens hyperparametre

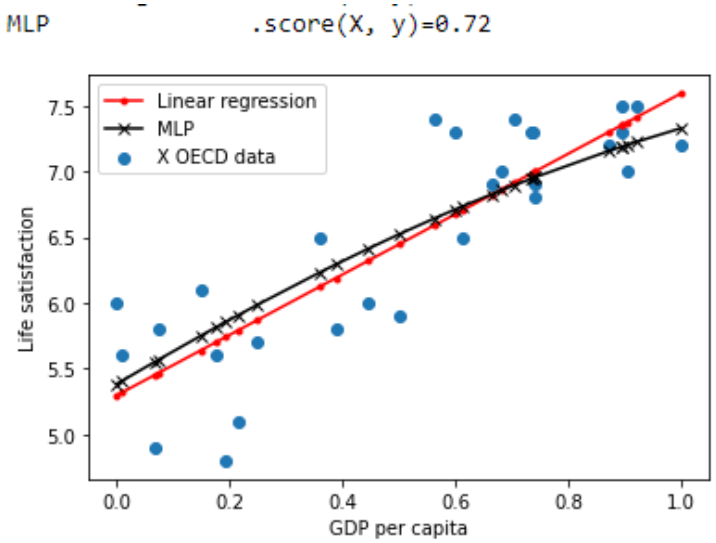
I denne opgave ønsker vi at ændre på hyperparametre for vores MLP-regressor, og se hvilken effekt de har på vores R^2-værdi.

Vi kan ændre antallet af neuroner ved at ændre parameteren hidden_layer_sizes, som er defineret i vores mlp-regressor. I de forrige opgaver har vi benyttet 10 neuroner. Ved tests er det fundet frem til, at en R^2-værdi på ca. 0.73 fås for alle neuroner ned til 10, undtagen når vi bruger 1 neuron. Når vi bruger 1 neuron, bliver vores R^2-værdi markant lavere, som betyder at regressionsmodellen ikke repræsenterer datapunkterne godt. Nedenfor på Figur 6 er dette vist.



Figur 6: mlp-regression med 1 neuron

Vi vil nu ændre på activation-funktionen til at bruge en sigmoid funktion, og solver-funktionen til at bruge, sgd, i mlp-regressoren og se om det ændrer på vores R^2-værdi. En test mlp-regression med 10 neuroner med disse ændringer er lavet, og er vist på Figur 7



Figur 7: mlp-regression med ny activation- og solver funktion

Ud fra testene ses det, at vi får en R^2-værdi på 0.72, altså meget tæt på de 0.73 som vi får med en ala aktiveringsfunktion. Man kan også se, at kurven er lidt kurvet, hvilket skyldes, at vi nu bruger en sigmoid funktion.

4 L05 - Linear regression 1

4.1 Qa - Lav en Python funktion, der bruger den lukkede form til at finde vægten w*

I denne opgave skal vægten w* udregnes i den lukkede form for et datasæt. Vægten er en værdi, der minimerer cost-funktionens totale cost.

For at kunne udregne vægten, så skal X-værdierne fra datasættet omskrives til en matrix-form bestående af to kolonner, hvor den første kolonne kun indeholder af 1-taller, mens den anden kolonne består af X-værdierne. Dette kaldes for biasing. Koden kan ses på Listing 10.

```
1 X1 = np.array([[8.34044009e-01],[1.44064899e+00],[2.28749635e
      -04],[6.04665145e-01]])
2 y1 = np.array([5.97396028, 7.24897834, 4.86609388, 3.51245674])
3 w1_expected = np.array([4.046879011698, 1.880121487278])
4
5 X1 = np.c_[np.ones((len(X1),1)),X1]
```

Listing 8: Omskrivning af X

Vægten kan nu udregnges ud fra følgende formel.

$$w\cdot = (X^T \cdot X)^{-1} \cdot X^T\cdot \tag{1}$$

På Listing 9 kan det ses, hvordan vægten udregnes. Da det er matricer der arbejdes med bruges .dot operatoren til at multiplicere og .inv operation til at invertere.

```
1 w = np.linalg.inv(X1.T.dot(X1)).dot(X1.T).dot(y1)
2
3 print(w)
4 print(w1_expected)
5
6 > [4.04687901 1.88012149]
7 > [4.04687901 1.88012149]
```

Listing 9: Udregning af vægten 'w' i den lukkede form samt test

Det kan ses, at den den udregnede vægt og den forventede vægt stemmer overens, som betyder at udregningen er korrekt.

4.1.1 Qb - Find grænserne for least-square metoden

I denne opgave skal den lukkede metode bruges til at udregne vægten 'w2' for et nyt datasæt med mere testdata. Samme formel som fra Qa bruges. Koden er vist på Listing 10

```
1  M=100
2  N=1
3  print(f'More test data, M={10}, N={N}...')
4
5  X2=2 * np.random.rand(M,N)
6  y2=4 + 3*X2 + np.random.randn(M,1)
7  y2=y2[:,0]
8
9  X2 = np.c_[np.ones((len(X2),1)),X2]
10
11 w2 = np.linalg.inv(X2.T.dot(X2)).dot(X2.T).dot(y2)
12
13 print(w2)
14
15 > [3.95879558  3.16661468]
```

Listing 10: Omskrivning af X

En af de ting man vil opleve er, at flere kolonner vil medføre større computeringstid. Dette skyldes de beregninger som inverse funktionen laver. Følgende formel beskriver computerings komplexiteten, hvor n er antallet af kolonner [Aurélien Géron, 2009].

$$O(n^{2.4}) \; til \; O(n^3) \tag{2}$$

I udregningen nedenfor kan man se, at en fordobling af kolonnerne i en matrix vil medføre, at computeringstiden bliver ca. mellem 5.3 til 8 gange større.

$$2^{2.4} = 5.3 \; til \; 2^3 = 8 \tag{3}$$

5 L05 - Gradient descent

5.1 Qa - Forklar Gradient descent algoritmen

Først bliver der genereret nogle datapunkter. Derefter bliver eta sat som er "learning curve". For loopet skal køres igennem 1000 gange med længden på 100. Derved får man en mere præcis "learning curve". Derefter bliver der lavet et 2x1 array med random tal. Til sidste køres forloopet igennem med alle de satte værdier og gradient descent formlen for at finde gradient next step. Se koden nedenfor på Figur 8.

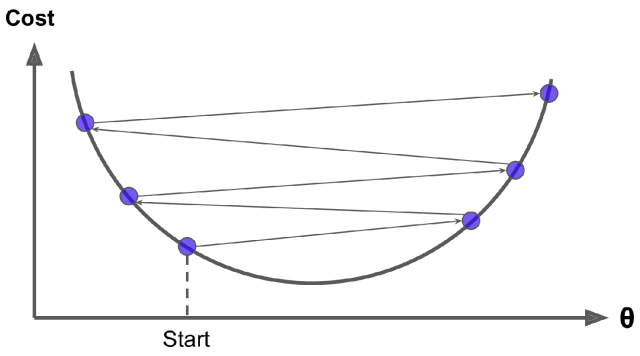
```
eta = 0.1  # learning rate
n_iterations = 1000
m = 100

theta = np.random.randn(2,1)  # random initialization

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

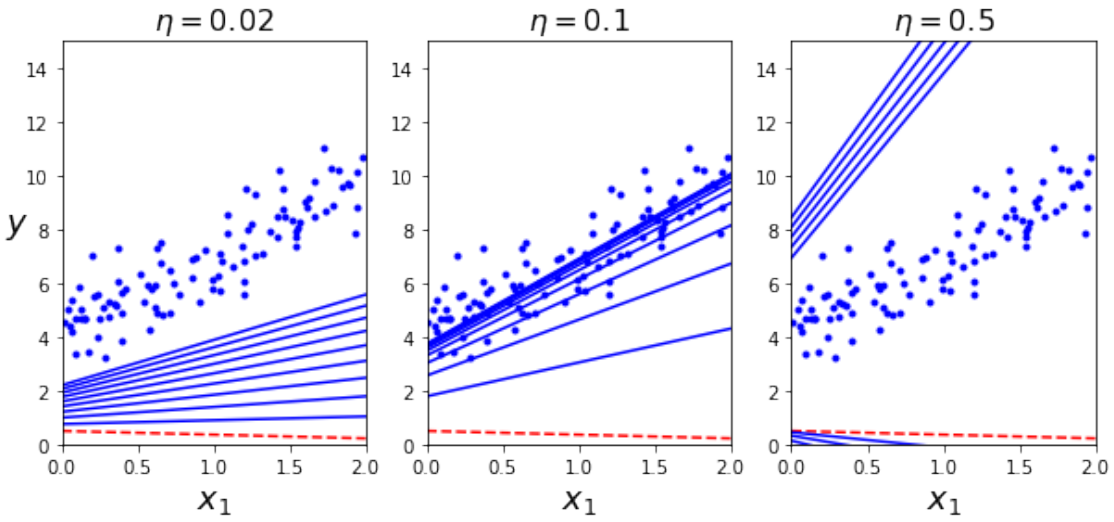
Figur 8: Gradient Descent kode eksempel

Når eta bliver ændret på er det mindre/større steps ift. genereret punkter. Heraf når eta bliver lavere tages der mindre steps mens hvis den bliver sat højere bliver der taget større steps. På nedenstående Figur 9 kan steps/spring ses med en høj eta.



Figur 9: Cost function med høj eta

Hvis man kigger på de 3 plots på Figur 10 som har henholdvis 0.02, 0.1 og 0.5 eta værdier, kan det ses at den bedste af de tre faktisk er 0.1 eta, da den passer bedst med de forskellige datapunkter. Mens 0.02 er under og 0.5 er langt over.



Figur 10: 3 plots med forskellige eta

5.2 Qb - Forklar og sammenlign Stochastic Gradient Descent (SGD) med GD

Den primære forskel er at GD tager hele datasettet mens SGD kun tager et datapunkt når den stepper. Dvs. GD er meget langsom ift. SGD når man har at gøre med store dataset. Dog er GD mere præcis end SGD fordi den tager hele datasettet. Når SGD er kørt færdig er værdien man får ud heller ikke optimal men tæt på. Når np.random.randint bliver kaldt for at lave et random array, hvor koden bruger disse værdier til at vælge et index. Når SGD formlen bliver kørt igennem. Koden kan findes under jupyter GITMAL/L05/gradient_descent under opgave Qb.

5.3 Qc - Forklar adaptive learning rate adaptive learning rate η med brug af SGD kode eksempel

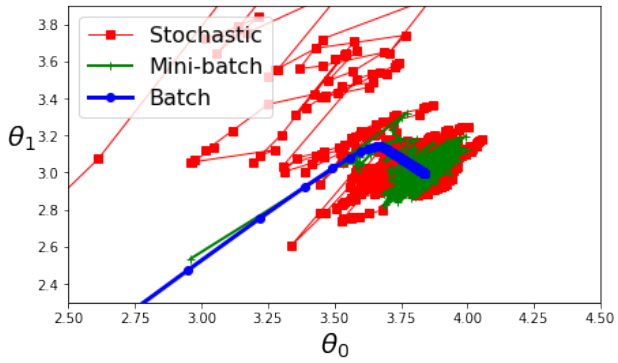
Ved at sætte learning rate kan man styre bedre hvordan SGD bliver brugt. Hvor man kan starte ud med at have en høj learning rate, og formindske den efterhånden for at komme ud af lokal optima. Derfor er det vigtigt ikke at formindske det for hurtigt, men heller ikke formindske det for langsomt. Idet man vil hoppe rundt om minimum ved en langsom learning rate. Learning rate for hver iteration kaldes også "learning schedule". Koden kan findes under jupyter GITMAL/L05/gradient_descent under opgave Qc.

5.4 Qd - Forklar og sammenlign minibatch method de to andre typer Gradient descent

Minibatch er lidt en kombination af hvordan der kigges på data fra de 2 andre. Den tager et set af random instances istedet for et ligesom SGD og hele datasettet som GD. Dette gør så den er langsommere end SGD, men stadig hurtigere end GD. Dvs. og med et større dataset rammer minibatch bedre minimum når den stopper end SGD, men langsommere.

5.5 Qe - Vælg en Gradient metode GD/SGD/mini-batch

Lige for denne for metode ligger valget helt klart på mini-batch idet at den følger GD rimelig godt også mens SGD starter et helt forkert sted. Dog kan det ses at der er meget mindre datapunkter for SGD så den kommer hurtigere ned omkring minimums værdien. Dog når den stopper er den ikke ligeså tæt på minimum som mini-batch og GD. GD er helt klart den mest præcise, men vil tage meget længere tid end både SGD og mini-batch. Så valget af metode afhænger af, hvor hurtigt algoritmen man udvikler skal processere data, og hvor præcis minimums værdien skal være.



Figur 11: Gradient Descent Methods

6 L06 - ANN

I denne opgave skal der laves et Overvåget Neuralt Netværk, kun igennem Scikit-learns. Med andre ord skal vi lave et 2-lags multi lags perceptron, igennem Scikit-learns MLPRegressor.

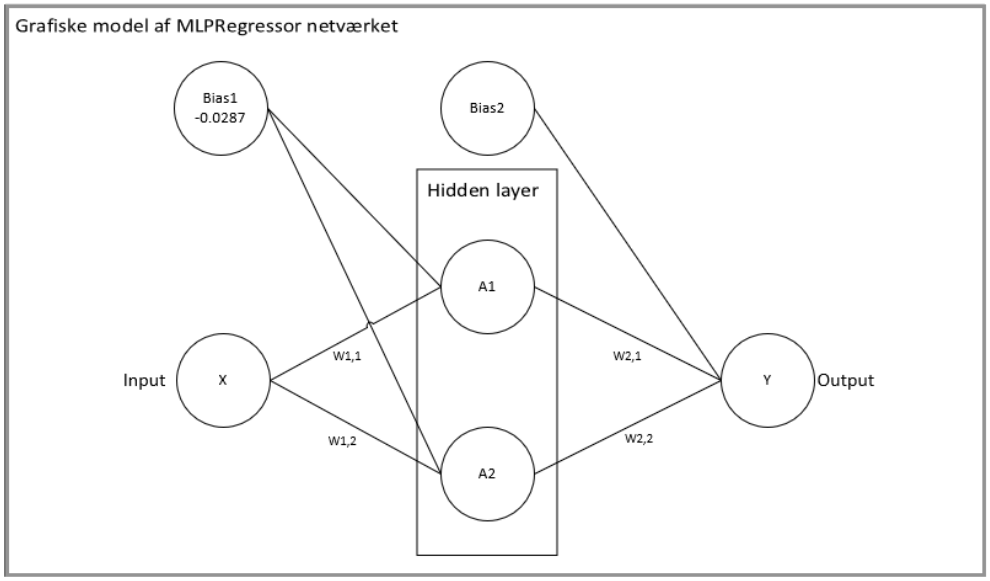
6.1 Qa: Fit modellen

Vi starter med at fitte vores model. Vi har fået givet noget data, i opgaven, dette data træner vi vores MLPRegressor model med.

```
1 X = np.linspace(-3,3,1000)
2 y = np.sinc(X)
3 X = X.reshape(-1,1)
4
5 mlp = MLPRegressor(activation = 'tanh', hidden_layer_sizes = 2, alpha = 1e
      -5, solver = 'lbfgs', max_iter = 1000, verbose = True)
6 mlp.fit(X, y)
7 Pred_mlp = mlp.predict(X)
```

6.2 Qb: Tegn en grafisk model

Der er lavet en grafisk model over netværket for at få en bedre forståelse for hvordan den virker.



Figur 12: Grafisk model af netværket

Som Figur 12 viser, så har vi et input i form af vores X, som er vores data. Vi har kun et hidden layer, med 2 led. Disse 2 led, er dem som essentielt laver udregningerne og sender det videre til outputtet. De 2 bias i toppen kan man se som nogle konstanter, som altid påvirker vores model. Disse eksistere ikke altid, medmindre man sætter dem, i vores tilfælde er bias 1 kun eksisterende.

6.3 Qc: Opskriv udtrykket

Udtrykket for vores model skal findes, vi kender den basiske funktion, men ikke de specifikke værdier. Så ved at bruge Coefs og Intercept, kan vi finde disse værdier.

```
print("Coef", mlp.coefs_)
print("Intercept", mlp.intercepts_)
> Coef [array([[ 3.54382464, -3.58327949]]), array([[-0.50619177],
       [-0.50608419]])]
  Intercept [array([-2.16720462, -2.19421214]), array([-0.02876178])]
```

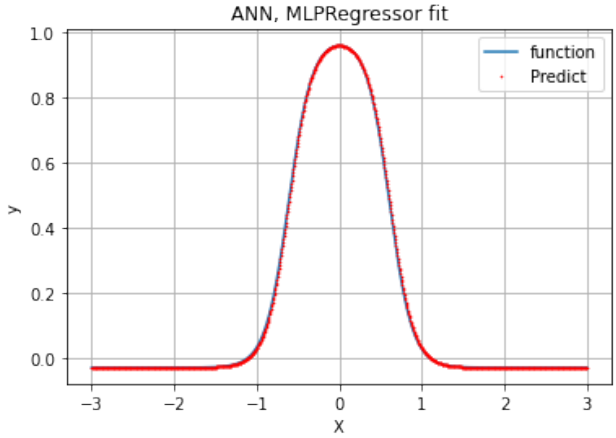
Vi får altså en masse værdier, som vi skal bruge i vores funktion, men problemet er hvilke tal der er hvad. Dog kan vi prøve os frem til vores funktion, indtil vi har et plot der ligner vores forventet plot. Efter forskellige test, er funktion blevet fundet til at være:

```
y = -0.506 * tanh(-3.52 * X - 2.16) - 0.506 * tanh(3.6 * X - 2.19) - 0.0287
```

Her er det nemmere at se hvad de individuelle tal er, f.eks. er det første tal i Coef arrayet og det første tal i Intercept arrayet er de første dele af funktionen.

6.4 Qd: Plot funktionen

Nu hvor vi har fundet funktionen kan vi plotte den.



Figur 13: Plot af funktion

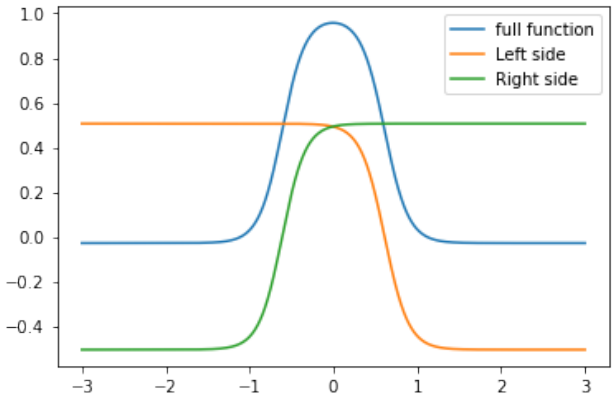
På Figur 13 ser vi, at vores funktion giver os et plot, der ligger perfekt oveni vores predicted plot. Dermed kan vi bekræfte at vores funktion er korrekt.

6.5 Qe: Plot første og anden del hver for sig

Her plotter vi den første og den sidste del af vores funktion altså:

```
1 −0.506 * tanh(−3.52 * X − 2.16)
2 − 0.506 * tanh(3.6 * X − 2.19)
```

Vi plotter de to dele og sammenligner dem med vores fulde funktion.



Figur 14: Plot af del funktioner

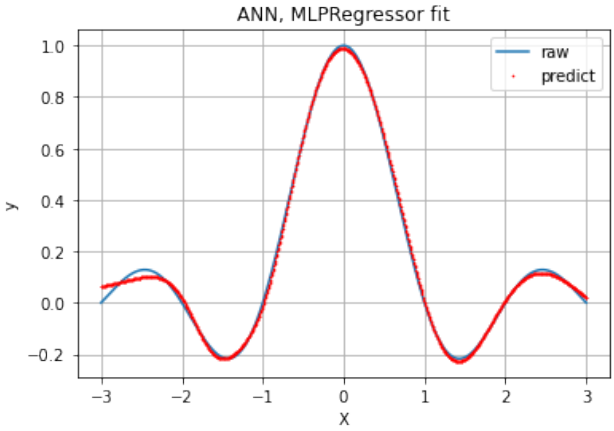
På Figur 14 ser vi de 2 dele af funktionen, henholdsvis gul og grøn. Den blå linje er vores fulde funktion, grunden til at den ikke ligger perfekt oveni, er på grund af det sidste tal der bliver trukket fra funktionen, som ikke er en del af de andre. Dette tal er bias-ledet. Kigger man på plottet, kan man tydeligt se, at de tilsammen skaber vores fulde funktion.

6.6 Qf: Fit funktionen med flere led

Vi øger mængden af led i vores hidden layer, så vi går fra 2 led til 5.

```
1 mlp = MLPRegressor(activation = 'tanh', hidden_layer_sizes = 5, alpha = 1e
      −5, solver = 'lbfgs', max_iter = 1000, verbose = True)
2
3 mlp.fit(X, y)
4
5 Pred_mlp = mlp.predict(X)
```

Det eneste vi ændre er altså værdien af vores hidden_layer_sizes til 5. Dette gør at vores model, har mere at arbejde med, derfor skulle dens resultater også blive bedre. Dette betyder dog ikke at man bare kan tilføje utrolig mange led til et lag, for at få bedre resultater, man bliver selv nød til at eksperimentere for at finde det bedste antal, det er altså en hyber parameter. Plotter vi vores nye model med 5 led i hidden layer, ser vi en betydelig ændring.



Figur 15: Plot af 5 led

Som Figur 15 viser så flader vores model ikke ud ved starten og slutningen, men formår næsten at have den helt samme form som vores predict.

Litteraturliste

[Aurélien Géron, 2009] Aurélien Géron (2009). Hands-on machine learning with scikit-learn, keras & tensorflow.

[Cortez et al., 2009] Cortez, P., Cerdeira, A., Almeida, F., Matos, T., and Reis, J. (2009). Modeling wine preferences by data mining from physicochemical properties. *Decision support systems*, 47(4):547–553.