Mercari Price Suggestion Challenge

Verslag leren & beslissen

Begeleidt door: Tim van Loenhout

N.L. de Bruijn, R. Kuiper, H.C. Otten & T. Ottens
Universiteit van Amsterdam

5 februari 2018

Samenvatting

In dit verslag wordt beschreven of het mogelijk is om prijzen te voorspellen op basis van producteigenschappen op Mercari. Trainingsdata bestaat uit zeven variabelen, inclusief prijs, per product, het doel is voorspellen van prijs op basis van de zes andere variabelen. Hier zijn stochastic gradient en gradient boosting decision tree modellen voor gebruikt. Na een looptijd van een uur levert dit een root mean square logarithmic error tussen de 0.42 en 0.45 op (240e plek uit 2045 op Kaggle op 3 februari 2018).

Introductie

Mercari is een online marktplaats die ook in de Verenigde Staten actief is. Voor de tweedehands spullen die op dit soort marktplaatsen verkocht worden kan het moeilijk zijn om een goede prijs te bepalen. De prijs hangt af van veel verschillende factoren. Mercari zou hun klanten graag willen helpen door aan de hand van reeds verkochte producten een prijs te voorspellen voor een nieuw aan te bieden product. Mercari wil deze nieuwe functionaliteit toevoegen aan hun site en om dat te bereiken hebben zij een wedstrijd gemaakt. Het doel van de wedstrijd is als volgt: voorspel aan de hand van informatie over een product de waarde van dat product. De wedstrijd wordt gehouden op Kaggle, een online platform dat vaker data science uitdagingen host. De uitdaging is te vinden op: www.kaggle.com/c/mercari-price-suggestion-challenge.

Mercari geeft de deelnemers van de wedstrijd, in overleg met Kaggle, de beschikking over een trainingset en een testset. Deze sets zijn beide comma seperated value (csv) bestanden en bestaan uit respectievelijk zeven en zes verschillende features. Bij de testset is de prijsfeature weggelaten, die moet voorspeld worden. Na het nader te ontwerpen algoritme moet een nieuw csv bestand op Kaggle ingeleverd worden, bestaand uit twee waardes een id, van het respectievelijke product uit de testset, en een voorspelde prijs.

Dit verslag beschrijft een poging de Mercari price suggestion challenge op te lossen: kunnen productprijzen worden voorspeld op basis van de andere zes eigenschappen?

Aanpak

Mercari heeft weinig informatie gegeven over succes in het verleden, of over hoe goed mensen dit probleem op kunnen lossen of iets dergelijks. Verschillende gebruikers op Kaggle hebben wel zogenaamde exploratory data analyses geplaatst met hun bevindingen tijdens het analyseren van de data.

Bij deze uitdaging was het verplicht om Python of R te gebruiken. Omdat wij allemaal gewend zijn om met Python te werken hebben wij daar voor gekozen. Op Kaggle werkt men in kernels, die komen in twee varianten: als notebook of als script. Een notebook bestaat uit meerdere cellen waardoor niet alles in een keer uitgevoerd hoeft te worden. Een script runt de hele code in een keer en is handig voor het behalen van een score.

Kaggle biedt de mogelijkheid om de kernels op hun site te draaien. Zij geven de deelnemers de beschikking over de volgende specificaties:

- 4 cores
- 16 GB RAM
- 60 minuten continue looptijd
- 1 GB aan opslagruimte.

In theorie is het voordeel van runnen op Kaggle dat er gelijkwaardige specs zijn voor iedereen. In praktijk blijkt Kaggle erg inconsequent, hoewel altijd langzaam. Dit bracht onzekerheid met zich mee en daarom hebben we de kleine aanpassingen lokaal uitgeprobeerd.

Voor deze uitdaging heeft Kaggle een trainingsset beschikbaar gesteld die bestond uit bijna anderhalf miljoen gelabelde samples. De data bevatte de volgende acht kolommen (met datatypes):

- train_id (int)
- name (object)
- item_condition_id (int)
- category_name (object)
- brand_name (object)
- price (float)
- shipping (int)
- item_description (object)

Daarnaast heeft Kaggle ook een testset beschikbaar gesteld met zeven kolommen, zonder price. De prijzen van deze set moesten voorspeld worden, en konden alleen door Kaggle worden beoordeelt.

Theorie

Algoritmen

We hebben gelabelde data, en willen een continue waarde (price, een float) voorspellen met een aantal verschillende soorten input variabelen. Als alle mogelijke waardes van deze variabelen worden omgezet in features die een bepaald artikel wel of niet bezit, kan een schaarse matrix worden gemaakt die compatibel is met veel soorten modellen. In deze fase van de Kaggle competitie zijn er al veel samples (1.482.535) en na preparatie ook veel features (zeker 50.000), in de volgende fase nog meer.

$$f(\mathbf{X}, \mathbf{w}) = \mathbf{w}^T \mathbf{X} \tag{1}$$

$$\mathbf{w} \leftarrow \mathbf{w} - \eta(\frac{\partial R(\mathbf{w})}{\partial \mathbf{w}} + \frac{\partial L(f(\mathbf{x}^i, \mathbf{w}), \mathbf{y}^i)}{\partial \mathbf{w}}) \tag{2}$$

$$E(\mathbf{w}, \mathbf{x}, y) = \frac{1}{n} \sum_{i=1}^{n} L(\mathbf{y}^i, f(\mathbf{x}^i, \mathbf{w})) + \alpha R(\mathbf{w}) \tag{3}$$

$$L(\mathbf{y}^i, f(\mathbf{x}^i, \mathbf{w})) = (\mathbf{y}^i - f(\mathbf{x}^i, \mathbf{w}))^2 \tag{4}$$

$$R(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{n} w_i^2 \tag{5}$$

$$\eta_t = \frac{\eta_0}{t^{\text{power\_t}}} \tag{6}$$

De trainingsset $\mathbf{X}$ bevat dan $(\mathbf{x}^1, y^1), ..., (\mathbf{x}^n, y^n)$ waarin iedere $x_j^i \in \{0, 1\}$ (genormaliseerd en heel schaars is), en iedere $y^i \in \mathbb{R}$. Dit is een regressie probleem, er moet een $\mathbf{w}$ worden gevonden waarmee formule 1 zo dicht mogelijk bij de werkelijke $\mathbf{y}$ van een arbitraire andere $\mathbf{X}$. Als het probleem niet lineair is moet $\mathbf{X}$ verder nog door een activatiefunctie.

Stochastic gradient descent (SGD) regressie lijkt daarom een passend model. Het is snel en gemakkelijk te gebruiken door implementatie in Sci-kit learn (b). In de documentatie staan een paar nadelen, namelijk sensitiviteit voor feature scaling en het vereisen van veel hyperparameters. Die zijn voor ons niet erg, respectievelijk omdat onze features al gescaled zijn en onze trainingsset niet variëerd.

Bij SGD wordt maar naar één sample (van een geshufflede trainingsset) gekeken bij het updaten van weights (Bottou, 2010), in tegenstelling tot de hele set of een batch zoals bij veel andere gradient descent algoritmen. Met

een paar onbekende termen die zometeen worden uitgelegd wordt de update met sample i gegeven door formule 2. Daarin is het nadeel van de Scikit-learn implementatie te zien: de error functie (formule 3) is niet zelf te definiëren.

Nu kunnen de L en R worden uitgelegd. Het zijn respectievelijk de loss en regularisatie functies. De gebruikte loss functie is squared error, gegeven in formule 4. De reden hiervoor is een gebrek aan keuze. Regularisatie wordt toegepast om de elementen van $\mathbf{w}$ laag te houden, en zo overfitting wat te verminderen. Er wordt hier L2 regularisatie gebruikt, gegeven in formule 6.

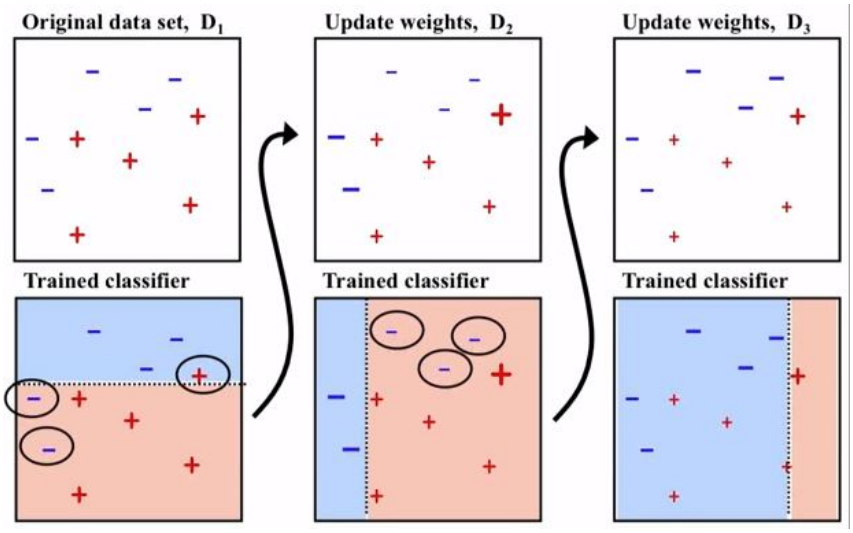
De laatste onbekende term, η, is de learning rate (stepsize). Deze bepaald hoe groot de verandering in gewicht per iteratie is. Deze is bij SGD relatief klein omdat er relatief veel iteraties zijn. Door de learning rate te verlagen naarmate meer tijd is verstreken heeft de SGD wat momentum. De methode heet inverse scaling, beschreven door formule 6 (met $0 <$ power_t ≤ 1 om de rate inderdaad te verlagen).

SGD is makkelijk te begrijpen en werkt redelijk goed. Maar er is na uitvoering nog wat tijd over voor een complexer model. Bijvoorbeeld ensemble methods, waarin meerdere modellen worden gecombineerd tot een. Als een model individueel bepaalde fouten maakt die andere modellen uit het stel niet maken, worden die fouten bij combinatie misschien niet meer gemaakt.

Wij maken gebruik van de boosting variant, waarbij er aan een klein model, een learner, een nieuwe learner wordt toegevoegd die de error verlaagd. Een groot voordeel van deze sequentie is dat nieuwe learners gespecialiseerd kunnen zijn in de fouten van vorige learners. In de error functie wordt bij boosting een weighted loss gebruikt, waarin $\mathbf{x}^i$'s die tot nu slecht gaan een zwaarder gewicht krijgen bij het bepalen van de volgende learner. Dit is voor classificatie weergegeven in figuur 1, waarin de belangrijke samples groter zijn weergegeven.

Wij maken specifieker gebruik van Microsoft's LightGBM implementatie van gradient boosting decision tree (GBDT) (Ke et al., 2017). Er worden continu kleine decision trees, stumps, gemaakt die de error op dat moment verminderen. Deze stumps groeien bij deze implementatie leaf-wise en kunnen afhankelijk van parameterkeuze dus relatief diep worden. Parameterkeuze is vrij belangrijk bij GBDT, en zeker met deze implementatie is veel mogelijk. Daarom in het aparte stuk hieronder meer.

De beste alternatieven op GBDT lijken random forests en long short term memory (LSTM) networks. De eerste is ook een ensemble method, een bootstrap aggregating in plaats van boosting variant. Er worden verschil-



Figuur 1: Enkele iteraties gradient boosting classificatie, uit Ensemble(4): Adaboost van Alexander Ihler op YouTube

lende normale decision trees gemaakt op delen van de data, de uiteindelijke voorspelling van het model is een gemiddelde van deze trees. De individuele trees hebben een relatief lage bias maar hoge variance. Bij GBDT hebben individuele trees juist een hoge bias en lage variance. Dit lijkt beter te passen bij onze grote hoeveelheid zeldzame features. Een LSTM network lijkt wel geschikt, maar training is relatief intensief, en het is moeilijk om in te zien hoe belangrijk bepaalde features zijn voor een uiteindelijke beslissing.

Analyse

Data analyse is grotendeels al bepaald door Kaggle; de voorspellingen van de testset worden beoordeeld met een root mean square logarithmic error, zie formule 7.

$$E'(\mathbf{w}, \mathbf{X}, y) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\log(f(\mathbf{x}^i, \mathbf{w}) + 1) - \log(y^i + 1))^2} \tag{7}$$

Omdat alle code voor iedere beoordeling door Kaggle moet runnen, is het beter ook lokaal te kunnen werken. Deze E' is in Python:

```
import numpy as np
def rmsle(f, y):
    d = np.log1p(f) - np.log1p(y)
    return np.sqrt(np.mean(np.power(d, 2)))
```

Omdat we lokaal maar één gelabelde trainingsset hebben, moet deze worden gesplit in een kleinere trainings- en een testset. (Dit om te voorkomen dat beoordeling plaatsvindt op dezelde data als waarop getraind is, een wel iets te grote beloning van overfitting.) Welke samples in welke set komen moet willekeurig zijn, hoewel onze trainingsset niet gesorteerd lijkt. Als een minder groot deel van de originele set in de trainingsset gaat, kunnen verbanden het model ontgaan, en met een minder grote testset zeggen de resultaten (hoeveelheid error op de testset) minder over de prestaties van het model. Er zal een split in train/test van 0,85/0,15 worden uitgevoerd. De testset lijkt klein maar is in absolute aantallen nog steeds groot. Bovendien moet de trainingsset voor sommige modellen, zoals LightGBM, nog een keer worden gesplit in een kleinere train- en validatieset voor validatie tijdens het trainen.

Een model wordt gefit op de trainingsset, waarna ermee wordt voorspeld wat de price van de samples in de testset (zonder kolom price) is. Deze voorspellingen gaan als `f` in de bovenstaande `rmsle` functie, de originele price kolom als `y`. Dit proces (inclusief split) kan herhaald worden om variantie in te schatten als cross validation, dit duurt alleen lang bij gradient boosting en is niet essentieel omdat de trainingsset zo groot is. Alles kan handmatig of door gebruik te maken van meer Scikit-learn functies.

Methode

Data preparatie

Om meer uit de geleverde data te kunnen halen hebben we per kolom de gegeven informatie onderzocht. We hebben een aantal aanpassingen gemaakt om er voor te zorgen dat de door ons gekozen algoritmes er zoveel mogelijk informatie uit konden halen. Een groot aanzienlijke hoeveelheid van deze opschoning werd uitgevoerd op kolommen die tekst bevatten. Bij het schoonmaken van deze kolommen voor het door ons geïmplementeerde algoritmen hebben we gebruik gemaakt van twee vectorizers van Sci-kit learn (a), namelijk: de countvectorizer en de term frequency inverse document frequency vectorizer (TFIDF).

Voor beide van deze twee vectorizers geldt dat ze de gebruiker de optie bieden om de input-tekst schoon te maken door het verwijderen van stopwoorden, leesteken en symbolen uit de tekst. Daarnaast zetten beide vectorizers de tekst in een kolom om in één grote schaarse matrix waarin wordt vermeld hoevaak ieder woord voorkomt in een iedere rij, met als grootste

verschil dat TFIDF gewichten aan de woorden toekend. Dit wordt gedaan vanwege het feit dat in grote verzamelingen tekst de kans groot is dat woorden als “sale” vaak voor komen zonder dat deze betekenisvolle informatie over de waarde van een product bevatten, deze woorden zouden dus een lagere waarde moeten krijgen. Dit wordt bij TFIDF gedaan door de frequentie waarin een woord voorkomt in een rij (term frequency) te vermenigvuldigen met het gewicht van dat woord dat wordt bepaald door middel van formule 8.

$$idf(t) = \log \frac{n_d}{df(d,t)} + 1 \tag{8}$$

$$v_{norm} = \frac{v}{||v||_2} = \frac{v}{\sqrt{v_1^2 + v_2^2 + ... + v_n^2}} \tag{9}$$

Hierbij is n_d het totaal aantal rijen in de set en $df(d,t)$ het aantal rijen waarin het woord t zich bevindt. Tot slot wordt, na de vermenigvuldigingen van de verschillende TF’s en IDF’s in een rij, de resulterende vector genormaliseerd door middel van euclidische normalisatie zodat elke vector een lengte van één heeft, zie formule 9.

Na het vergelijken van de behaalde resultaten met beide vectorizers bleek voor ons de TFIDF betere resultaten op te leveren en hebben we ervoor gekozen deze te gebruiken tijdens het prepareren van de name en item_description kolommen. Hieronder zullen we per gebruikte kolom een overzicht geven van het preparatie proces.

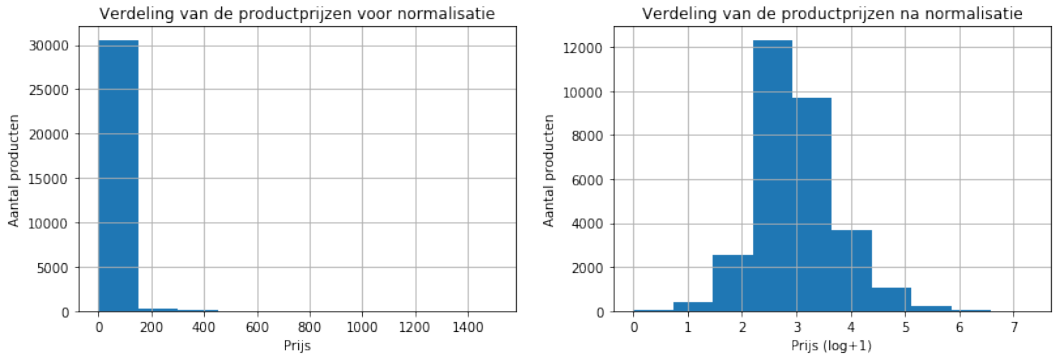
De name kolom bevat de door de verkoper aangegeven titel van het product en bestaat dus uit stukken tekst. Echter bevat de kolom naast relevante tekst ook veel extra tekens die schijnbaar door de gebruiker worden gebruikt om hun product titel te decoreren. Het opschonen van deze tekst is gedaan met behulp van hierboven genoemde TFIDF. We hebben er in deze kolom voor gekozen de stopwoorden te negeren, woorden die minder dan vijftien keer voorkomen verwijderd en combinaties van woorden ook te zien als mogelijke feature. Dat laatste, een ngram range van meer dan 1, wordt gedaan om combinaties als “Galaxy S6” te herkennen.

De item_condition_id kolom bevat voor ieder product een geheel getal tussen één en vijf dat aangeeft in welke staat het betreffende product is. Deze enkele kolom hebben we omgezet in vijf losse kolommen die per mogelijke waarde aangeven of het product in die staat is. Dit betekend dat voor ieder product vier van deze vijf kolommen een 0 is en één kolom een 1. Dit is gedaan om de categorie compatibel te maken met de rest van de schaarse

matrix waarin er per rij wel of geen feature van toepassing is. Daarnaast hebben we onderzocht of er producten waren waar de name of item_description suggereerde dat het product nieuw was terwijl de item condition iets anders zei, dit bleek slechts in 0,2% van de rijen het geval te zijn. Omdat de operatie die dit zou corrigeren tijd kost en het percentage zo laag was hebben we besloten dit niet in ons algoritme mee te nemen.

De category_name kolom bevat weer strings. Er waren een aantal rijen die nog geen categorie string hadden, en gevuld moesten worden omdat not a number (NaN) waarden later in het proces problemen opleveren. De waardes van de category_name bestonden uit strings die het volgende patroon volgden: “hoofdcategorie/subcategorie/subsubcategorie”, we hebben er voor gekozen deze categorieën op te splitsen in drie kolommen van de drie losse categorieën. Dit is gedaan omdat bijvoorbeeld hoofdcategorieën misschien al iets zeggen over de prijs. Zonder split zou deze correlatie bij alleen al een andere subsubcategorie niet gebruikt kunnen worden. De resulterende strings worden in features omgezet met behulp van de countvectorizer van Sci-kit learn (a). Deze maakt simpelweg een woordenboek van alle (sub)categorieën en zet deze woorden om in features die een sample wel of niet bezit.

Ook de brand_name kolom bevatte strings, en missende waarden (NaN’s) moesten vervangen worden om string operaties uit te kunnen voeren. Om de kolom om te zetten in features is ieder merk gezien als aparte feature, waarbij we er voor hebben gekozen alleen de 3.000 populairste merken als feature te zien, hier hebben we voor gekozen omdat de merken die niet bij die 3.000 horen vaak slechts enkele malen voor komen in de set waardoor we kunnen voorspellen voor die merknamen. Het omzetten van de merken in features hebben we gedaan met de label binarizer van Sci-kit learn (a) omdat deze simpel in gebruik is en de optie biedt de minder populaire merken te negeren. In totaal zijn er meer dan 600.000 NaN’s in de trainingsset. Daarnaast bleek dat in 150.000 van de samples waarin de kolom nog niet was ingevuld, de merknaam wel werd genoemd in de name en/of item_description kolommen.



Figuur 2: De verdeling van de productprijzen voor (links) en na (rechts) de normalisatie van de prijzen

Om de brand_name kolom completer te maken hebben is geprobeerd de lege merknaamkolommen aangevuld door voor de honderd populairste merken te kijken of deze in de name of item_description van het product voor kwamen en deze indien dat het geval was in te vullen als brand_name van het product. Dit had alleen een negatief effect op de behaalde error. Dit negatief effect kan verklaard worden door het incorrect uitlezen van merknamen uit de beschrijving en de naam. Onze manier om de merken uit de overige informatie te halen is als volgt: er werd gekeken, voor de 1.000 populairste merken, de merknamen, als los woord, voorkwamen in de beschrijving of naam. Omdat de brand_name feature niet een significant effect had op de uiteindelijke score hebben wij besloten om de missende waardes in te vullen met een niet bestaand merk.

Voor de shipping kolom, die bestond uit een waarde van 0 of 1 hebben we onderzocht of er producten waren opgegeven waarvoor deze kolom zei dat het niet nieuw was terwijl de description en name kolom wel de string “brand new” bevatten, dit bleek in minder dan 5% van de rijen wat waardoor corrigeren geen significant effect had op onze error score.

Tot slot kwamen we er bij het analyseren van de prijzen achter dat deze skewed waren naar links zoals te zien in figuur 2 . Om deze afwijking te verminderen hebben we ervoor gekozen de prijs te normaliseren door het natuurlijke logartime + 1 van de prijs te nemen. + 1 om ook met price 0 om te kunnen gaan. De verdeling van de prijzen voor en na deze operatie is weergegeven in figuur 2

Algoritme optimalisatie

Voor de SGDRegressor van Scikit-learn zijn de volgende parameters gebruikt:

```
alpha: 0.0001
average: False
epsilon: 0.1
eta0: 0.01
fit_intercept: True
l1_ratio: 0
learning_rate: "invscaling"
loss: "squared_loss"
max_iter: 420
penalty: "l2"
power_t: 0.25
random_state: 42
shuffle: True
tol: None
warm_start: False
```

De loss functie is zoals gezegd niet perfect, Kaggle gebruikt namelijk iets anders.

De GBDT van LightGBM heeft meer parameters, een volledige lijst is te vinden in de documentatie op https://lightgbm.readthedocs.io/. De parameters die voor deze applicatie interessant bleken zijn:

```
test_split_size = 0.12 # eerder genoemde validation set size
num_boost_round = 6000 # meer is beter, pas aan voor
                       # ongeveer 59 minuten op Kaggle
early_stopping_rounds = 500 # niet relevant op Kaggle zelf

learning_rate: 0.7,
application: "regression",
boosting: "gbdt"
max_depth: 4,
num_leaves: 60,
metric: "RMSE", # root mean square, dus zonder ln
bagging_fraction: 0.6
```

De belangrijkste aanpassingen zijn de `learning_rate`, die veel groter is dan de standaard 0.1, en `max_depth`, die standaard ∞ is en hier dus relatief

laag. Deze aanpassingen maken het fitten van het model allebei een stuk sneller, en bleken empirisch geen groot negatief effect te hebben op de behaalde score. Hetzelfde geldt voor `bagging_fraction`, deze parameter geeft de fractie van de data aan die voor iedere iteratie wordt gebruikt.

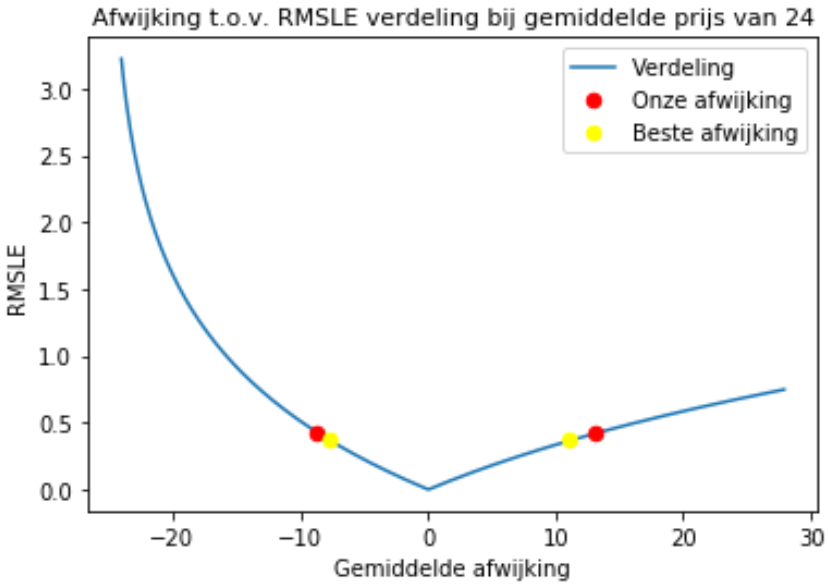
De verschillende algoritmen worden met elkaar gecombineerd door de voorspellingen van de SGD regressor vermenigvuldig met 0.4 op te tellen bij de voorspellingen van de GBDT met 0.6. De GBDT blijkt namelijk iets beter te werken, hoewel de voorspellingen in het algemeen volgens een t-test niet significant ($p < 0.05$) verschillen.

Resultaten

Na het runnen van de code moest een csv bestand gemaakt worden dat bestond uit de id van het te voorspellen product en de voorspelde prijs. Dit bestand werdt door Kaggle gecheckt en een score werd gegeven. De uiteindelijke score verschilt nogal maar dit komt door de inconsistente van Kaggle.

Uiteindelijke score

- Op Kaggle's testset een RMSLE tussen de 0,425 en de 0,45;
- De gemiddelde prijs was gemiddeld afgerond $24;
- Op 3 februari was dit nog goed voor een 240e plek (van 2045) op het leaderboard op Kaggle.



Figuur 3: Alternatieve weergave van verschillende resultaten op Kaggle

Analyse

Bruikbaarheid resultaat

De behaalde score (RMSLE) ligt dus tussen de 0,42 en 0,45. Als wij dit vergelijken met de tot nu toe best behaalde score op Kaggle op 3 februari, 0.37, komen er meerdere vragen bovendrijven. De gemiddelde te voorspellen prijs is ongeveer $24. Bij dit gemiddelde leveren de genoemde RMSLE scores, als ieder punt dezelfde afwijking zou hebben, de afwijkingen in figuur 3 op.

Deze grafiek kan gemakkelijk verwarren. Onze score levert twee gemiddelde afwijkingen op. Door de manier waarop RMSLE werkt is het lager inschatten van een prijs veel kostbaarder voor de uiteindelijke score. Dit betekent dat in het geval er voor elke prijsschatting er onderschat is, de gemiddelde afwijking in ons geval 9 dollar is. Is elke prijsschatting echter overschat dan is de gemiddelde afwijking in ons geval 13 dollar. Onze gemiddelde afwijking ligt dus ergens tussen de 9 en 13 dollar, de hoeveelheid over-en onderschatte producten bepaalt waar precies.

Als wij vervolgens onze gemiddelde afwijkings range vergelijken met die van de beste score, tussen de 8 en 11 dollar, dan vragen wij ons af of het uiteindelijke resultaat wel bruikbaar is. Een gemiddelde afwijking van ongeveer 9, voor de beste score, op een gemiddelde prijs van ongeveer 24 is zo veel dat de voorspelde prijzen waarschijnlijk in veel gevallen niet representatief

zullen zijn voor de echte waarde van het product.

Dit betekent echter niet dat alle voorspelde prijzen ernaast zitten. Mogelijkerwijze zit een aanzienlijk gedeelte zodanig dichtbij de echte prijs dat het wel de moeite waard is om het aan de klant te presenteren. Dit lijkt het geval te zijn bij populaire producten. Een mogelijke toevoeging is om een betrouwbaarheid van de voorspelde prijs aan de klant te presenteren. Zo kan de klant zelf bepalen wat het met de verkregen informatie besluit te doen. Dit kan bijvoorbeeld door een prijsrange gegeven worden in plaats van een absolute prijs. Deze aanpassingen geven de klant naar onze mening een completer idee van de waarde van zijn product.

Oplosbaarheid

Mercari zal zelf bepalen of de price suggestion challenge een bruikbaar resultaat op heeft geleverd. Wij vragen ons echter af of het probleem gesteld door Mercari met de huidige mogelijkheden überhaupt oplosbaar is. Voordat we kunnen stellen of het probleem oplosbaar is moeten we ons eerst afvragen wanneer het probleem opgelost is.

Naar ons weten heeft Mercari deze wedstrijd uitgeschreven om een nieuwe functie te implementeren voor hun site. Deze functie houdt in dat klanten een voorspelde prijs voorgeschoteld krijgen als zij een nieuw product aan willen bieden. Naar alle waarschijnlijkheid is het probleem voor Mercari dus opgelost als zij deze functie uit kunnen rollen. Voor Mercari is het dus belangrijk dat een oplossing voor dit probleem ook bruikbaar is. Naar ons idee dragen de door ons voorgestelde veranderingen, prijsrange en betrouwbaarheid, bij aan de bruikbaarheid. Maar dat betekent niet dat het probleem dan al opgelost is. Door de manier waarop RMSLE werkt kan het zijn dat je een lage score hebt zonder ook maar een enkele prijs goed te schatten. Dit betekent dat alle voorspelde prijzen ongeveer in de buurt zitten van de echte prijs. Maar een lage score kan ook bereikt worden door een gedeelte zo goed als perfect te voorspellen en de rest significant slechter te voorspellen. Het lijkt ons van belang voor Mercari dat er met een hoge recall voorspeld wordt, dat de precision daar onder lijd kan worden verzacht door de door ons voorgestelde aanpassingen. Door een range te geven in plaats van een absolute prijs, en een betrouwbaarheid mee te geven, kan de klant zelf bepalen hoeveel gebruik van de service te maken. De eigenlijke prijs van tweedehands producten is vaak toch subjectief.

Incomplete data

Het oplossen en/of bruikbaar maken van een oplossing wordt verder bemoeilijkt door het feit dat gebruikers/verkopers de informatie over het product verstrekken. De volledigheid van de data hangt af van de verkoper. Deze verkoper kan te weinig data invoeren, kan liegen over het product, of heeft misschien een product aangeboden voor een onredelijk hoge of lage prijs. Met andere woorden: de ingevoerde data is verre van betrouwbaar. Dit maakt de kans op een relevante voorspelling kleiner.

Conclusie

Wij concluderen dat hoewel onze algoritmen een relatief goede oplossing zijn, de prijs van producten niet goed genoeg te voorspellen is. De data zelf is namelijk niet betrouwbaar en compleet genoeg om elke combinatie aan eigenschappen om te zetten in een prijs. Wij raden Mercari daarom aan de opzet van deze wedstrijd lichtelijk te veranderen zodat de focus ligt op het teruggeven van een betrouwbaarheid naast de voorspelde prijs. Na deze veranderingen helpt het resultaat de klant daadwerkelijk bij het bepalen van een prijs, hoewel misschien door aan te raden gewoon zelf te zoeken.

Referenties

Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer.

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T.-Y. (2017). Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, pages 3149–3157.

Sci-kit learn. Feature selection. `http://scikit-learn.org/stable/modules/classes.html#module-sklearn.feature_extraction.text`. Bekeken op 4 februari 2018.

Sci-kit learn. Stochastic gradient descent. `http://scikit-learn.org/stable/modules/sgd.html`. Bekeken op 4 februari 2018.