Methodology:

I first created a dataset in Excel, with ten users. Nine have salaries randomly chosen between $400 and $600 per week. I picked numbers for the last user, for an example described below. Each user has eleven rows of data. The first entry for each user (week = 0) is an average of however many weeks of historical pay data we have for that user.

I save the hypothetical data as a csv file, and open it in python. I first calculate simple variables: salary after the fee, historical average after fee, and the difference between the two.

This difference is the building block for calculating what users get in pay ("Pay") and "Balance in App." The floor (limit) on the balance a user can have – one week's salary of debt – requires us to use modified differences. I do this in the following order:

1. Calculate the running total (accumulation sum) of the difference in actual and average salary for each user each week, using a floor so that this value is never greater than one week's salary.
2. Flag instances where users are at the limit of borrowing.
3. Calculate the difference in balances (from step 1) from week to week.
4. Recalculate the difference between actual and average salary – the building block – using the flags and the difference in balances from week to week:
    a. Where the user can borrow the full difference between that week's salary and average pay, take the unmodified difference.
    b. Where the user can borrow a limited amount (which takes them up to the max) or none at all, take the difference in balances from week to week, from step 3.
5. Recalculate the running total, which is the "Balance in App," using modified differences from step 4.
6. Lastly, calculate "Pay," equal to:
    a. Average pay when a user is not at the borrowing limit.
    b. Salary plus credit available with the app when a user reaches the borrowing limit that week.
    c. Salary when the user has reached the borrowing limit.

The lending model produces two key variables: Balance in App and Pay. Here's a snapshot for one user, who reached the borrowing limit in Week 7:

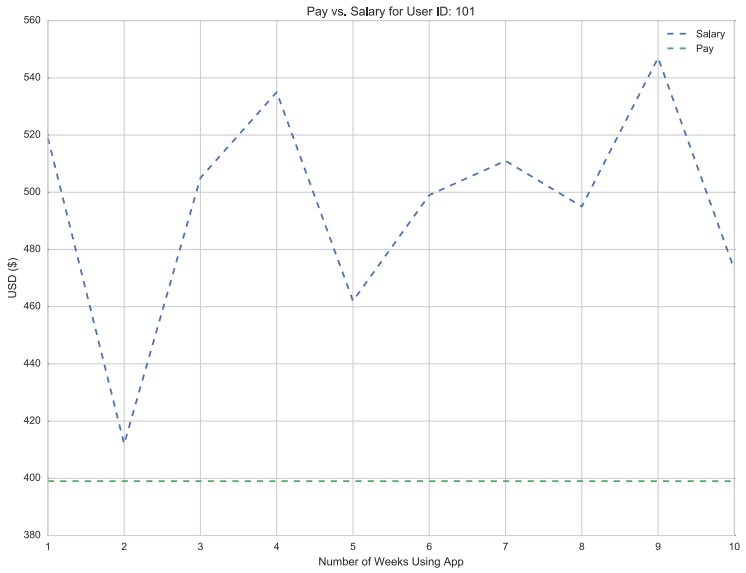
User ID	Week	Salary	Historical Average	Salary After Fee	Flag	Balance in App	Pay
104	1	$544	$579	$541	0	-$38	$579
104	2	$503	$579	$500	0	-$117	$579
104	3	$476	$579	$473	0	-$223	$579
104	4	$432	$579	$429	0	-$373	$579
104	5	$596	$579	$593	0	-$359	$579
104	6	$495	$579	$492	0	-$446	$579

104	7	$427	$579	$424	1	-$579	$557
104	8	$518	$579	$515	1	-$579	$515
104	9	$466	$579	$463	1	-$579	$463
104	10	$479	$579	$476	1	-$579	$476

If the user were to leave after using the app for ten weeks, he or she would be in default.

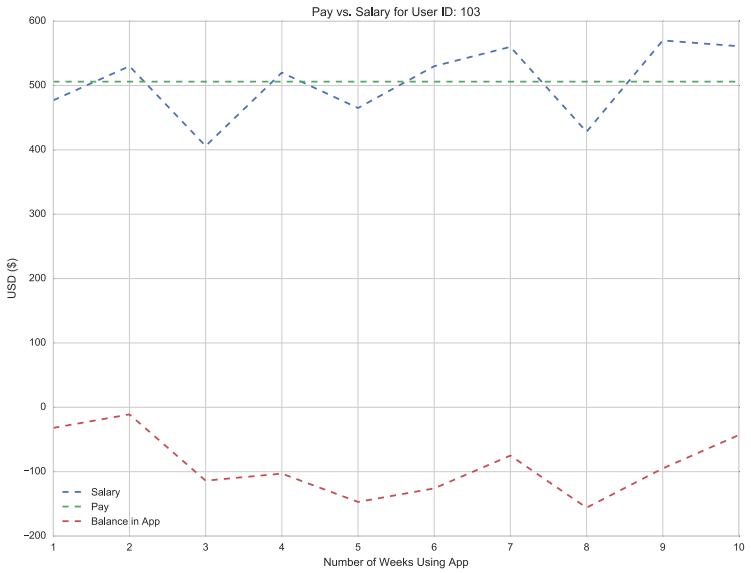
Understanding the app's Impact:

Users who get the app and experience a rise in pay save more. For example, this user never gets a boost, and saves the difference between the lines for Salary and Pay:

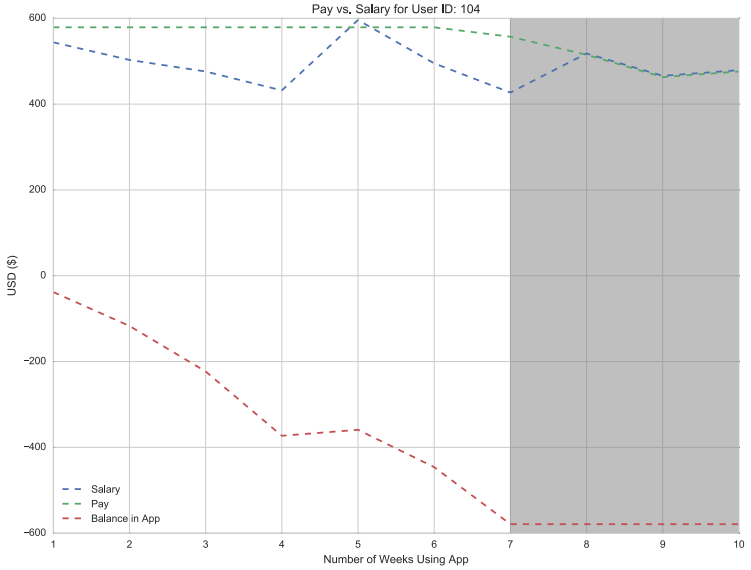


Users who experience such a rise might want to update the level of average pay in steps, which could be easily done using a rolling mean.

Other users aren't able to save as much, but still benefit from having smoother incomes:



Here's an example of a user who experiences a decline in weekly salary, reaching the borrowing limit at Week 7.



We also know that if the Pay accurately reflects average income, then the app will be providing a cushion to its users approximately half the time, or twice a month. If a payday loan costs $15 for every $100 borrowed per week, and the app is able to help users avoid fees associated with $200 of week-long payday loans twice per month, the app saves users $48 per month after accounting for the app's fees. (Math: 2*2*15 – 4*3 = 48)

We can also measure the impact on users in aggregate:

In this dataset (with historical salaries and average salaries that are randomly generated), by week ten, users have saved on average 16.7% of a week's salary.

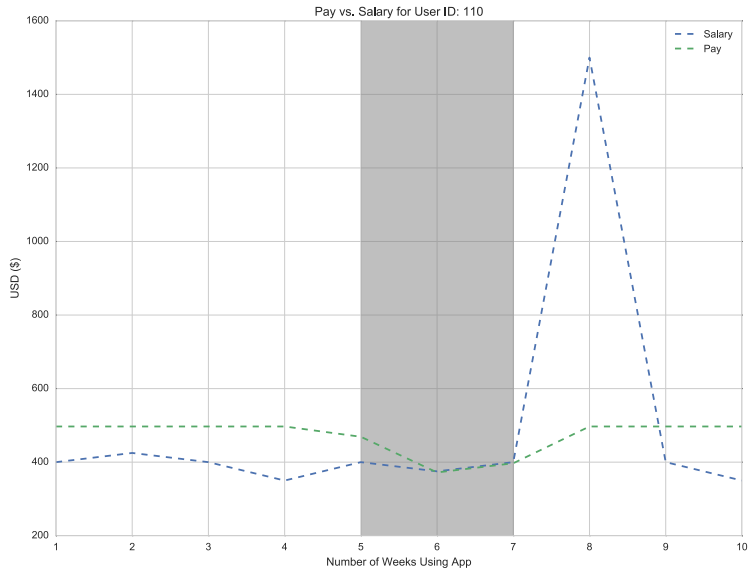
We could compare the average standard deviations in historical pay and in Pay (which is zero for users who never reach the lending max, and very small for users who do) to show how the app solves a big problem for many hourly workers.

What rates, constants, and factors are most important?

- The level of **Pay:** if it's the average of historical pay, then savings will be positive only if average salary during use of the app is higher than the average of historical pay.

- **Savings Rate** (or one-time contributions)**:** users who save even just by 5% or 10%, by setting a lower Pay, are less likely to reach the app's lending maximum, and will have more stable pay for a greater number of weeks if they experience periods of low salaries.

- The **distribution of a user's salary** affects the benefit of using the app. I picked numbers for the tenth user to show how Pay should be modified to account for uneven patterns in pay. The user experiences consecutive below average paychecks, and one bumper paycheck that is three times the average. Updating the average pay, once a trend of below average paychecks is established, would decrease the rate at which the user approaches the lending maximum. Here's a plot of the data:



The user's average historical and average salary while using the app is $500. The gray area indicates the weeks in which the user was at the limit (beginning at week 5).

- The **borrowing limit**, the level at which the app won't lend anymore to a user, affects the app's impact. In the case of the user plotted above, a more flexible floor would help the user avoid payday loans or fees before receiving an above average salary in week 8.

Notes on my model:

A richer model would account for more user information when setting the amount of Pay. Information regarding job role, minimum hours worked per week by law, tips, seasonality, whether a user commutes to work, price trends in expenses like food and energy, and one-off events in historical pay data could be valuable when setting and adjusting the Pay to decrease volatility in incomes, help users pay bills, and incentivize saving.