

Detection of Diabetic Retinopathy using Machine Learning

CS 476

Vaibhav Sharma, Will Wallace



Fall 2019

1. Problem Definition

Diabetes is one of the fastest-growing epidemics worldwide. Here at home, Saskatchewan has experienced a 58% jump in type 2 diabetes.^[1] A report from 2017 estimates that 1 in 10 people in Saskatchewan are currently living with the disease. Another 14% of the total population are considered to be in a prediabetic state (at risk of developing the disease).^[2]

One of the primary indicators of diabetes is the presence of diabetic retinopathy (DR). This is a condition affecting the eye and one of the primary causes of blindness.

Being able to identify those who are at risk of developing diabetes is an important task, but this also means having to screen tens of thousands of Saskatchewan patients in an efficient manner (or tens of millions worldwide).

Complications to Consider:

- Not everyone who has diabetes (or is in a prediabetic state) may be aware of it.
- Similarly, not everyone who has diabetic retinopathy is aware that their eyes are being damaged. The warning signs may not be noticed until it is too late, at which point vision loss begins to accrue, leading to blindness. Patients with DR may believe their eyes are fine because they have not noticed any vision disturbances yet.
- Not everyone has access to a clinic where they can be seen quickly for an appointment.
- It is recommended that diabetic patients have an annual eye exam, but clinics are often over-booked and sometimes patients do not show up at all.

From a healthcare point of view, the problem is to screen a large number of patients in a short period of time. However, ophthalmologists can only visit with one patient at a time.

1. There is often a large imbalance between the number of diabetic patients and clinicians who can provide care, e.g. thousands of patients per specialist.
2. Clinicians are professionally trained, but are still human and can make mistakes.
3. As well, clinicians are prone to fatigue and stress after working long hours.

From a software engineering point of view, the problem is to design a system that can accurately and efficiently diagnose diabetic retinopathy by analyzing uploaded images of a patient's eye and indicate to the clinician whether there are signs of disease present. The goal would **not** be to replace trained specialists, but to provide them with a useful tool to make their life easier and reduce the chance of making mistakes when diagnosing patients.

[1] <https://www.cbc.ca/news/canada/saskatchewan/diabetes-rates-saskatchewan-climb-1.3933762>

[2]

https://www.diabetes.ca/DiabetesCanadaWebsite/media/About-Diabetes/Diabetes%20Charter/2018-Backgroundunder-Saskatchewan_JK_AB-edited-13-March-2018.pdf

2. Feasibility Study

Currently there exist numerous convolutional neural networks (CNNs) which attempt to diagnose diabetic retinopathy, each with varying levels of accuracy.

Examples of preexisting projects are easily found:

An extensive collection of preexisting projects can be found at kaggle.com, which hosted a 2015 competition with a prize pool of \$100,000 for the most accurate systems. The competitors' notebooks are located at <https://www.kaggle.com/c/diabetic-retinopathy-detection/notebooks>, as well as in their respective GitHub repositories.

- **Benefit:** As far as feasibility is concerned, many of these projects openly offer their code for study or reuse, ensuring that we do not find ourselves in a position where an entire model has to be written and trained from scratch, which can require days of computation time.
- Thanks to the numerous available projects containing pre-trained models, our project becomes much more feasible.
- In particular, we will be adapting code from Kaggle user **tanlikesmath** and their ResNet50 Oversampling project:
<https://www.kaggle.com/tanlikesmath/diabetic-retinopathy-with-resnet50-oversampling>
- As well, we will be studying a project by GitHub user **tmabraham** and their SqueezeNet CNN project:
<https://github.com/tmabraham/retinopathy-classifier>

Greater Automation:

Existing projects, such as those listed above, require more steps from the user before their image is uploaded and analyzed. While we will adapt code from them, we also want to reduce the amount of buttons that the user has to click to ensure a smoother operation.

As well, we will introduce a brand new feature to continually upload more images without the user having to manually start over; the page will constantly be ready for the next patient's images.

Benefit: Our interface will have a simpler design and more automation. This means more user friendliness and efficiency; images will be automatically assessed without the user having to repeatedly click 'Submit' or 'Clear' to start over.

High accuracy is ensured:

As will be noted in Part 3(c) of the report, correctness is our primary concern.

Numerous projects exist with accuracy ratings above 80% or even 90%. Many of these projects were created as entries in the 2015 Kaggle competition linked above.

Benefit: By employing a pre-trained model, we will ensure our project enjoys similar rates of accuracy.

Preexisting datasets are easily accessible:

As mentioned above, we would like to avoid training new models from scratch, as this is computationally expensive and very time-consuming. Depending on the size of the dataset and the hardware being used, training a model can take not just hours, but days. However, if we need to train or re-train a model, there are freely accessible datasets to help achieve this.

- For example, [Kaggle](#) hosts an 82 GB dataset consisting of ~35,000 high resolution retinal images.
- The images provided to Kaggle were originally provided by [EyePACS](#). Each image has been graded by a professionally trained clinician on a five-point scale (0 – 4), with 0 indicating a healthy eye and 4 indicating severe proliferative retinopathy.
- As it pertains to our project, the grades given to each image can be used as a set of training labels so the neural network can learn to distinguish between healthy and diseased eyes. Most datasets, including the EyePACS dataset, already include these training labels in a separate .csv file.
- **Benefit:** If we find ourselves in a position where training or re-training becomes necessary, the feasibility of our project will not be affected thanks to the copious quantities of training data available.

Greater speed:

We have noticed that greater speeds may potentially be demonstrated by our Django application when compared to the Flask application used by tmabraham's GitHub linked above.

Although Flask is more lightweight compared to Django, **we will avoid using the networked threading that was employed by tmabraham's app, as this type of threading can be a significant source of slowdown.**

Increasing worldwide demand and overall benefits:

As noted in the problem definition, diabetes is an epidemic which continues to grow worldwide. As a result, the need for timely intervention and efficient diagnosis-making is more dire than ever. While there are numerous pre-trained models in existence, they are not easily accessible to healthcare providers and are mostly only well-known to computer scientists!

The overall benefit of our project is to combine accurate pre-trained neural network models on the back-end with a web-based, more user-friendly, fully automated, and easily accessible front-end to provide efficient screening for diabetic retinopathy to physicians anywhere in the world.

3. Software Requirements Elicitation and Specification

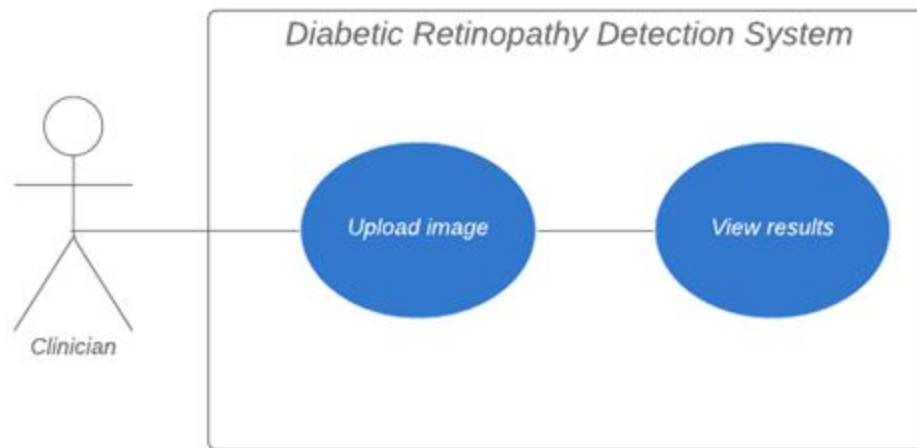
(a) Functional Requirements

- **Upload an image:** Allow a clinician (e.g. an ophthalmologist) to send a retinal image from their local hard drive or shared network drive to be examined by a pre-trained deep learning model.
- **Display results:** After successfully uploading their image, the user is presented with the results of the analysis:
 1. Probability that disease is present (e.g. an output of 0.91 would indicate a 91% chance of the patient being afflicted with diabetic retinopathy).
 2. Probability that disease is not present.

(b) Use Case Diagrams, Use Case Descriptions, and Activity Diagrams

Note:

- **As discussed with Samira earlier in the semester**, there will typically be only one realistic use case for the “real-world” deployment of this software.
- A professionally trained clinician, such as an ophthalmologist or other technician, will be responsible for using the software in their office; **it is a medical tool designed for a single purpose by a specific type of end user.**
- Therefore, there are no other potential end users, and only a single use case description is necessary.



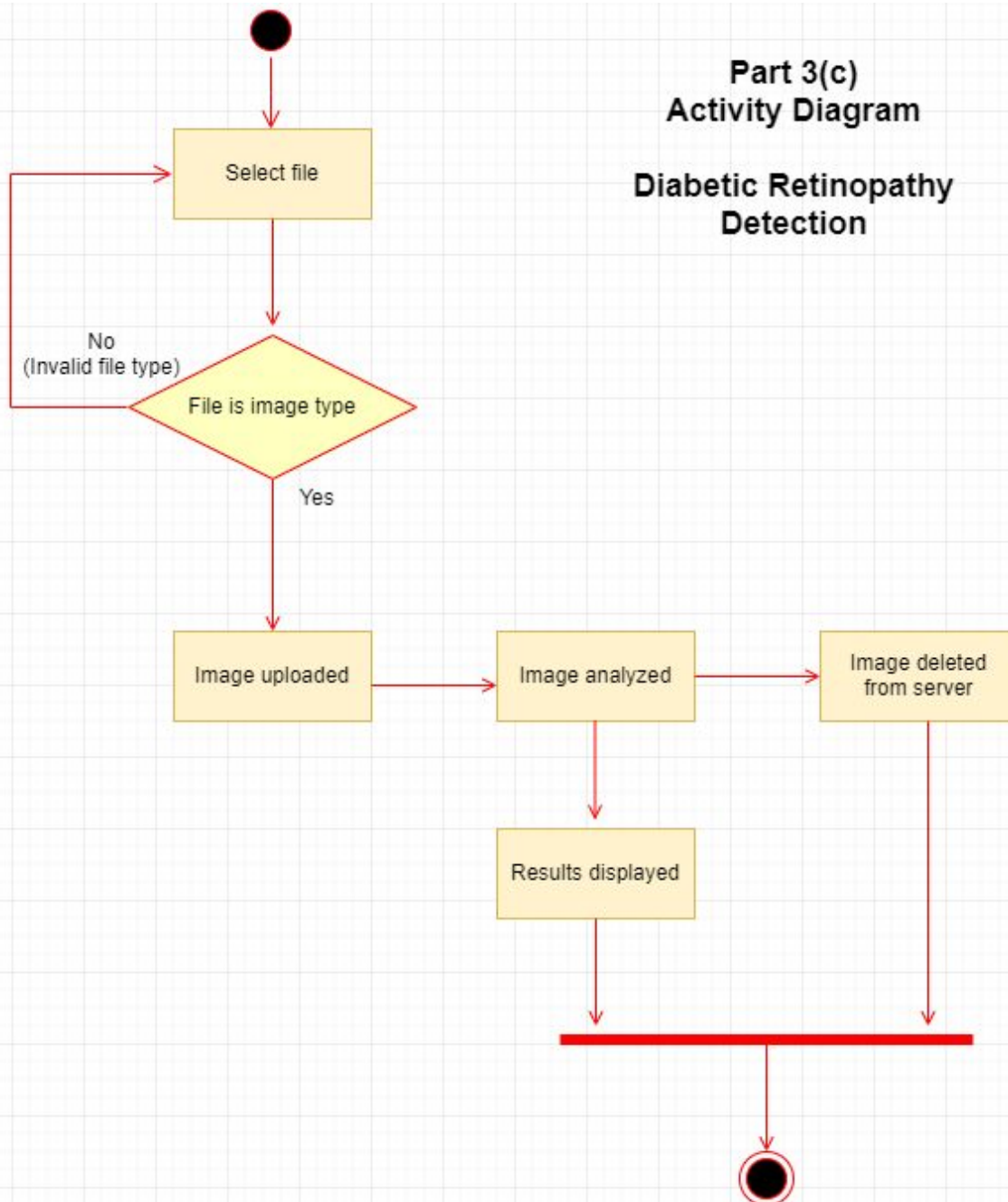
Use Case Description:

Use Case 1 Summary	Goal: Ophthalmologist uploads image and views the diagnostic results.
Example Actor	Ophthalmologist or other healthcare professional in a busy eye-care centre.
Trigger Precondition	There is a backlog of images that have been taken from patients, but none of the images have actually been examined yet.
Description of step 1	Ophthalmologist accesses web app either by clicking icon or entering URL.
Description of step 2	Ophthalmologist selects image which is locally stored either on the physician's hard drive or on the clinic's shared network drive.
Description of step 3	Ophthalmologist uploads image.

Description of step 4

Ophthalmologist views results (and determines the next step of treatment for the patient; however, this is beyond the scope of our project).

Part 3(c)
Activity Diagram
Diabetic Retinopathy
Detection



(c) Software Qualities

Correctness:

The healthcare and medical science application domain is a field of software engineering where correctness is paramount. Why? Although software engineers involved in medical science are not the physicians who provide treatment directly to patients, the tools and systems designed by software engineers may have a very real impact on patients' health.

Therefore, if a system is designed which provides inaccurate results, the well-being of patients may be jeopardized.

Consider the impact of a false-negative, in which the patient is told they are healthy and discharged from hospital, only for the mistake to be noticed later when the disease has progressed to an unmanageable state. This can occur, for example, in cancer screening when an early-stage tumor is missed and therefore allowed to grow unnoticed.

Alternatively, consider the mental and emotional stress a patient might experience when given a false-positive. They may believe they have developed a terminal disease which will soon cost them their life (for example, if using another system which detects cancer), or they may otherwise worry about upcoming surgeries, consultations, treatments, and financial costs – only to be told later that it was a mistake.

For the aforementioned reasons, **we are considering correctness to be the most important software quality of our project.**

Radiologists, ophthalmologists, and other physicians have traditionally been responsible for manually examining all forms of diagnostic imaging (e.g. x-rays, ultrasounds, CTs, MRIs, nuclear medicine bone scans, and – most pertinent to this project – funduscope retinal imaging and ophthalmoscopic screening).

These are highly trained professionals who have passed through medical residency and continued on to specialize in their preferred fields. Some will have decades of experience in visually identifying disease. Any software which is intended to help them must be extremely accurate in its output.

Time Efficiency:

Although correctness is our primary concern, time efficiency is also extraordinarily important in a healthcare setting.

While it is never acceptable to “rush” towards a diagnosis, we do have to consider the importance of providing results in a timely manner.

Consider that ophthalmologists are often outnumbered by their patients by hundreds or even thousands. For example, in Thailand, it is estimated that 5,000,000 diabetic patients are served by less than 1,400 ophthalmologists. This means that each specialist is responsible for an average of 3,500 diabetic patients, and this does not include anyone else needing eye care.

As mentioned earlier in the report, a doctor can only meet with one patient at a time, and doctors are human; they can make mistakes, and this is compounded by factors like fatigue and stress.

By ensuring that our system provides quick results without sacrificing correctness, we can increase the number of patients served per day. By being time-efficient, our software can increase the productivity of a clinic while simultaneously decreasing stress and fatigue on specialists who need to be clear-headed in their decision-making.

User Friendliness:

As mentioned above, our typical end user is expected to be a trained clinician such as an ophthalmologist or other specialist working in an eye-care clinic. They should not be expected to be an expert in working with computers and should never be left in a position to just “figure it out” on their own. Specialists in healthcare have little free-time on the job and cannot waste any portion of their day trying to navigate a clumsy or confusing design.

Following from this, our project will adhere to the commonly accepted heuristics of usability. Chief among these, as far as our project is concerned, is a minimalist and aesthetic design. Simplicity is key. **With no unnecessary “bells and whistles” attached, user interaction will be more focused, straightforward, and productive.**

The interface should never leave the user wondering what they are capable of doing or where they are, so navigation will be kept as simple as possible.

Leaving aside the graphical appearance of the app, we can also consider its user friendliness from a logical point of view: How easy is it to learn? Is there a steep learning curve, or can a relatively new user easily pick up the product and begin using it without worry? By keeping concerns like these in mind, we will ensure that user friendliness is maintained at all stages of execution.

We have aimed for full automation with as few buttons as possible, to make the experience of our app as smooth and easy as possible for our users.

Robustness:

Our typical user is expected to be a medical professional such as an ophthalmologist or a technician working in an eye-screening clinic. They should not be expected to be computer experts and should not be required to do extensive trouble-shooting on their own.

Therefore, the system should be able to recover from errors or, preferably, prevent errors from arising at all.

It should be fault-tolerant and be prepared with the consideration that a user may accidentally attempt to upload something that is not an image. For example, a physician in a hurry might inadvertently select an MP3 file for upload.

In a high-volume setting with many patients scheduled for appointments every day, an unexpected software failure can be disastrous; patients will show up for their appointment, only to be told that they cannot be seen until the problem has been fixed. Appointments would have to be cancelled and rescheduled, and tempers would flare.

In other words, a lack of robustness will impact not only our software's performance but would also potentially affect the well-being of patients.

For this reason, and with the expectation that the software will be employed in a busy clinic or hospital, robustness is a crucial software quality for our project.

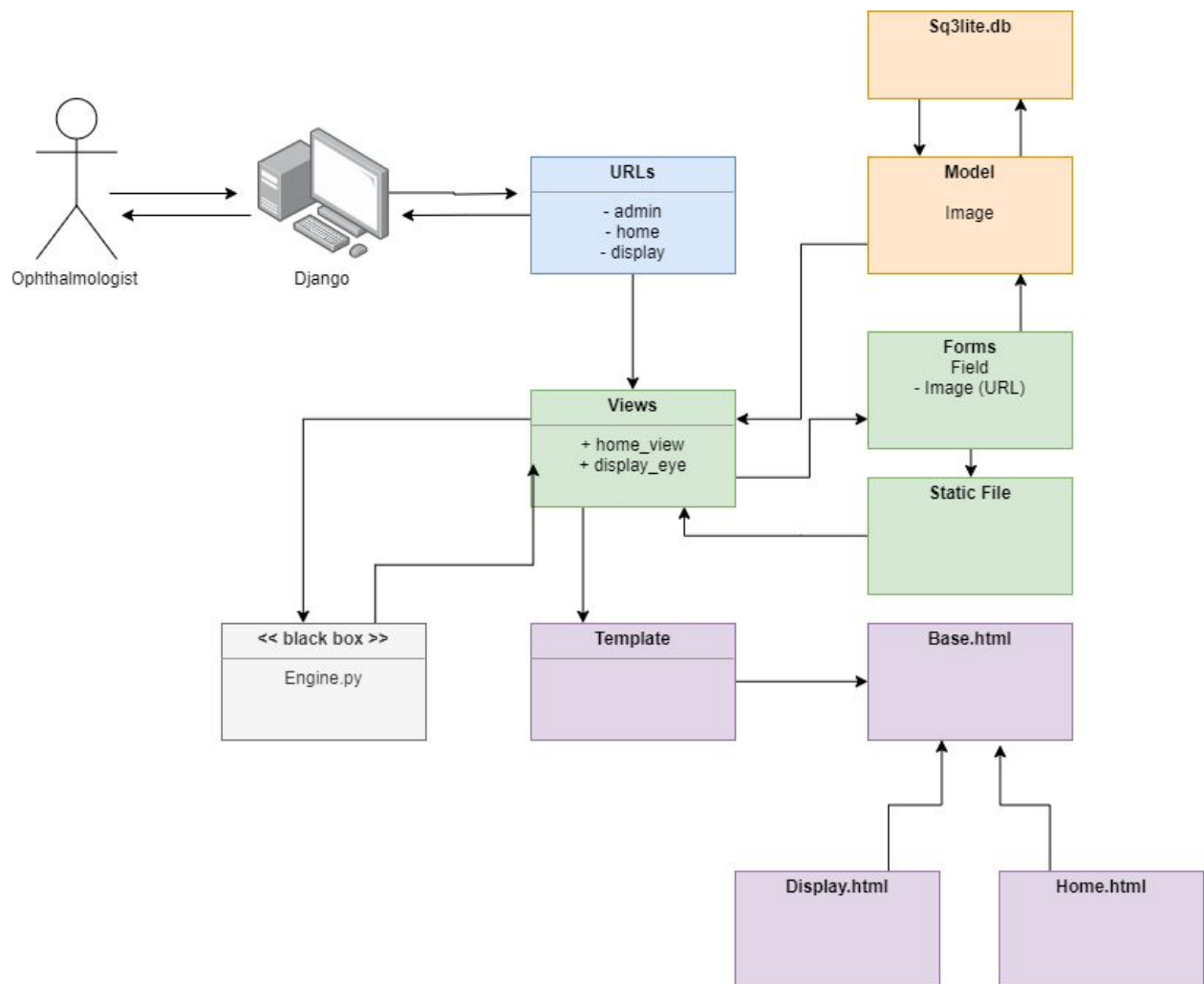
4. Top-level and Low-level Design

(a) Logical Software Architecture: MVT (Django's version of MVC)

For our software architecture we used Django's variant of MVC, which is named MVT (model - view - template). Neither of us have worked with MVT before, but we were both familiar with the more traditional MVC.

Django's version of MVC is quite similar, but the role of the view is changed. The view in MVT plays more of a controller role, and the display (HTML) is handled by the template.

A simple look at Django's MVT architecture:

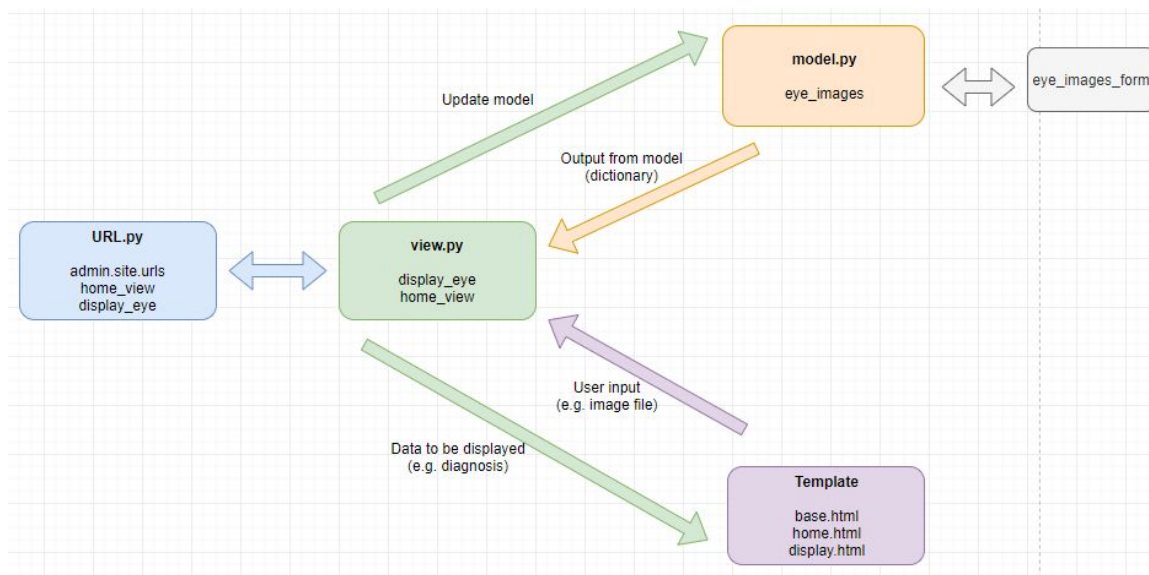


A few things to note:

- The “URL” depicted above is *not* the address the user accesses in their web browser. This is a URL which is internal to Django.

- **The view has taken a more forward role in Django** and is no longer “behind the scenes” as in traditional MVC.
- If you would like to learn more, here is a link to the official documentation from Django regarding their MVT architecture: [MVT and MVC](#)

MVT Software Architecture for Diabetic Retinopathy Detection:



We chose to implement our project with MVT software architecture because it is essentially built directly into Django.

It is perfect for use in Python projects, upon which Django is based.

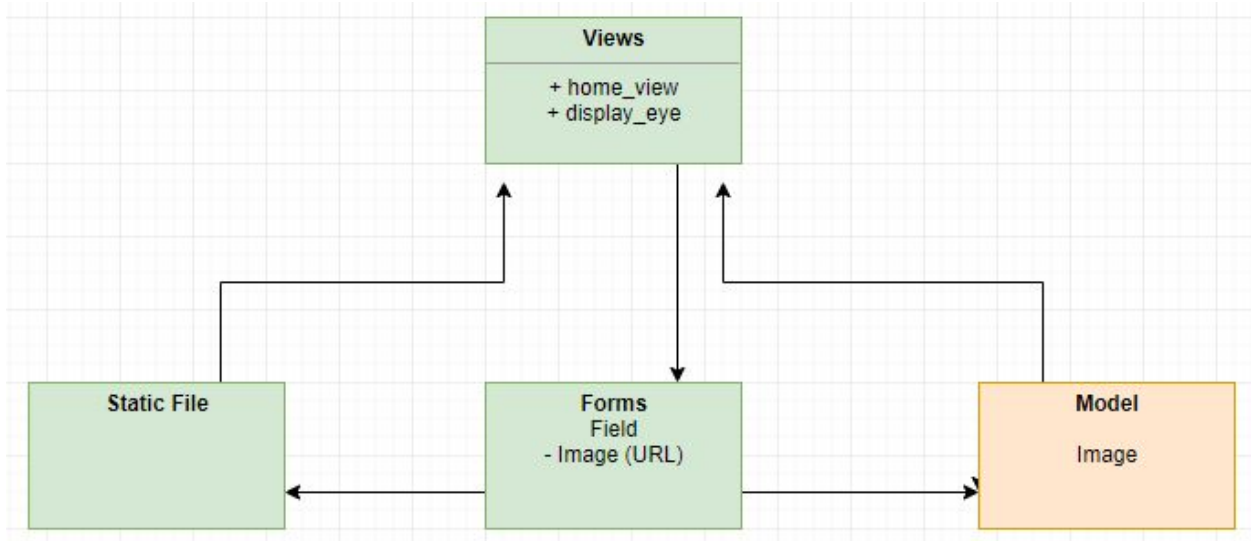
MVT is also especially good for artificial intelligence projects, by allowing us to keep the trained model(s) isolated; only the view (which is now acting as the controller in MVT) is allowed to interact with the model.

There are two lower level design patterns which were implemented in this project

- MVT
- Simple Factory

Model view Template as explained above allowed us to make this project highly Cohesion. MVT was implemented on low level between Views , Forms and Models.

The image below shows how MVT is implemented on low level design.



In this low level architecture, a form object is initiated by the view as it can be seen in the image below.

```
def display_eye(request, my_id):

    var = {}
    if request.method == "POST":
        form = eye_images_form(request.POST, request.FILES)
        if form.is_valid():
            new_form = form.save()
            my_id = str(new_form.id)
            form = eye_images_form()
            return redirect('/display/' + my_id)
```

Here the view uses display_eye function to create a form object. This form object includes an image field which is added via frontend. This form takes all the data from the user that is supposed to be saved as an object in the database and create a form object which can be inserted in the database. The reason forms are used to create an object rather than directly creating an object in model is because forms help clean the user input which assist in security and prevents data/model corruption to inconsistency in the database.

```
class eye_images_form(forms.ModelForm):

    class Meta:
        model = eye_images
        fields = [
            'image',
        ]
```

Once the object is created and saved in the model, it is then retrieved by the view on GET request as shown in the figure below.

```
obj = get_object_or_404(eye_images, id = my_id)
var["image"] = obj.image
var["file_upload"] = True
var["uploaded"] = ""
var["obj"] = obj

image_result = predict_result(str(obj.image))

var["resultObj"] = image_result
```

This design pattern helps maintaining low coupling and high cohesion as all the components interacting in this design pattern are completely independent and changes in one will not have regressive effect on the other.

Another reason to choose MVT for low level design is to maintain consistency between high level and low level architecture as we are using Python Django framework which itself is designed using model view template design.

model.py

```
class eye_images(models.Model):
    image = models.ImageField(upload_to= 'datasetsImages/')

    def get_absolute_url(self):
        return reverse("model_detail", kwargs={"pk": self.pk})
```

forms.py

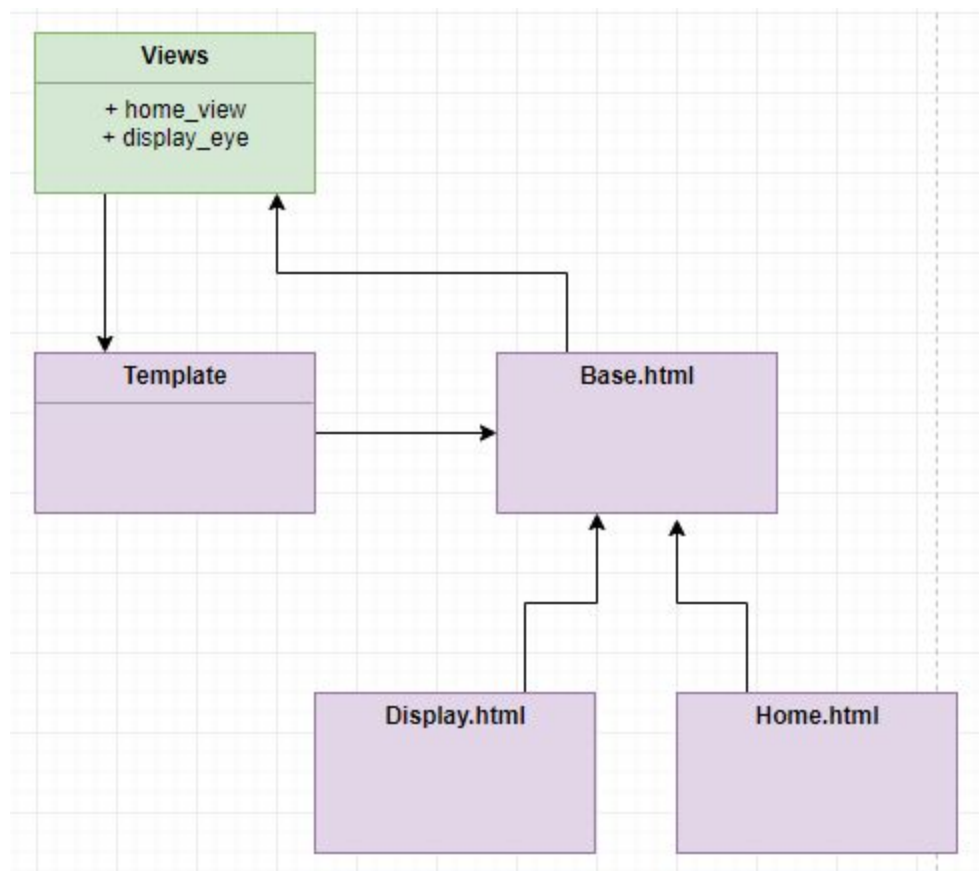
```
class eye_images_form(forms.ModelForm):

    class Meta:
        model = eye_images
        fields = [
            'image',
        ]
```

One of the key features of using MVT on low level design is efficiency. Forms allow user data to be saved as a url in a dictionary instead of an image itself. Image is never stored in the Model or the Form

but instead the URL(local location of the image. Not https) is saved and moved around the different functions. When user uploads the image to be processed, Django Forms automatically saves it a static directory names as " datasetsImages" under the source directory. This allows our architecture to be highly efficient on both high and low level as it puts an end to heavy local and network threads. One of our goals found during the feasibility study was to make this application more efficient than the one that exist already and this design pattern allows our Django application to achieve that.

Simple Factory method was used in html inheritance and to create objects based on parameters. This low level architecture is implemented between base.html, home.html and display.html
Base.html is main html file which have django blocks which decides the final result.



Snippet from home.html

```
{%block section_id%}
  <section id="about">
{%endblock%}
```

Snippet from display.html

```
{%block section_id%}
  <section id="Result">
{%endblock%}
```

Snippet from base.html

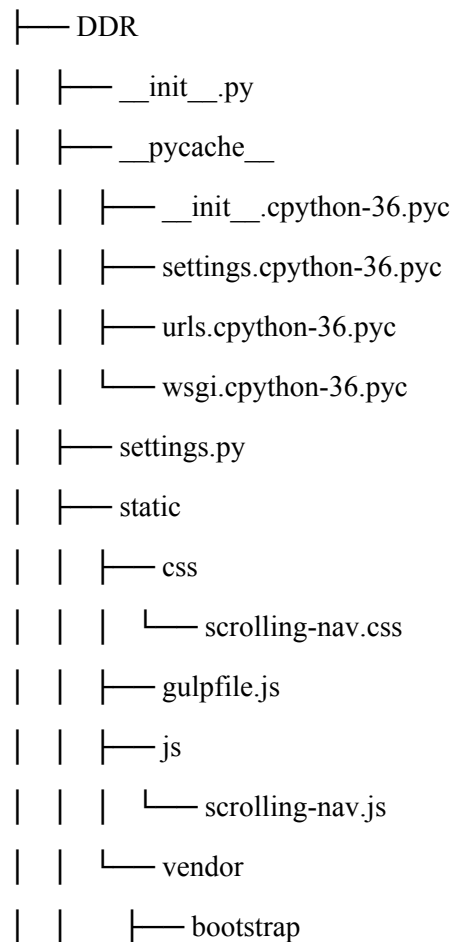
```
{%block section_id%}
{%endblock%}
<div class="container">
  <div class="row">
    <div class="col-lg-8 mx-auto">
      {%block about_result%}
        <h2>Diabetic Retinopathy </h2>
        <p class="lead">Diabetic retinopathy (DR), also known as diabetic eye disease, is a med.
        <p class="lead">This website detects Referrable Diabetic Retinopathy (RDR) using Deep Lea
      {% endblock %}
    </div>
  </div>
</div>
</div>
</section>
```

As shows in the snippets attached from all the different html files, it's an example of how different call from views.py is creating different html files. When view requesting page rendering on either display.html or home.html, the whole process of page creation is completely abstracted from the client(view) requesting the model.

This design patterns helps maintain the consistency along all the html pages and adding new html files does not affect any other pre-existing htmls. Hence, result in high cohesion and low coupling.

Part 5. Programs

(a) Component Diagram (created using draw.io):



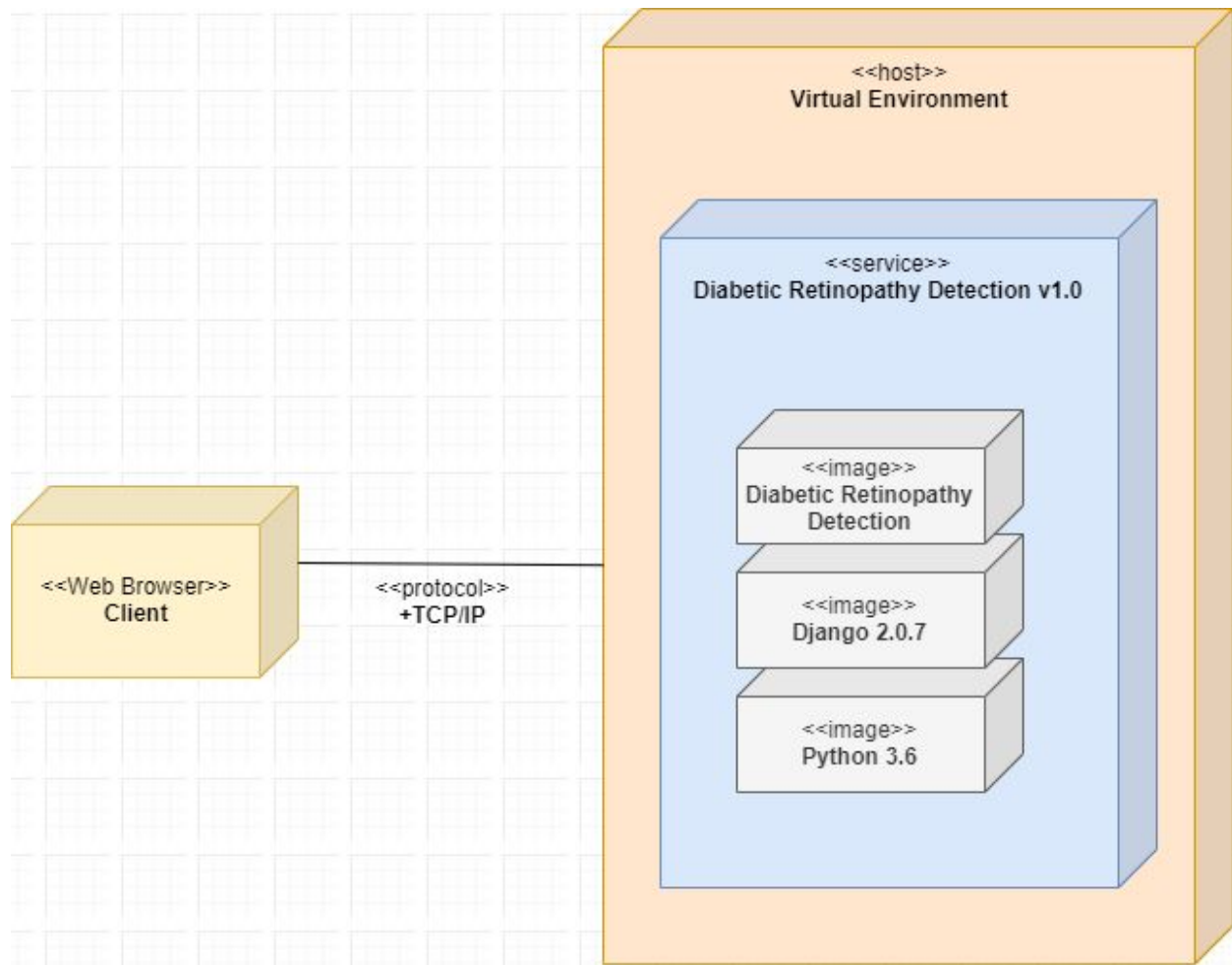
- └─ css
 - └─ bootstrap-grid.css
 - └─ bootstrap-grid.css.map
 - └─ bootstrap-grid.min.css
 - └─ bootstrap-grid.min.css.map
 - └─ bootstrap-reboot.css
 - └─ bootstrap-reboot.css.map
 - └─ bootstrap-reboot.min.css
 - └─ bootstrap-reboot.min.css.map
 - └─ bootstrap.css
 - └─ bootstrap.css.map
 - └─ bootstrap.min.css
 - └─ bootstrap.min.css.map
 - └─ js
 - └─ bootstrap.bundle.js
 - └─ bootstrap.bundle.js.map
 - └─ bootstrap.bundle.min.js
 - └─ bootstrap.bundle.min.js.map
 - └─ bootstrap.js
 - └─ bootstrap.js.map
 - └─ bootstrap.min.js
 - └─ bootstrap.min.js.map
 - └─ jquery
 - └─ jquery.js
 - └─ jquery.min.js
 - └─ jquery.min.map
 - └─ jquery.slim.js
 - └─ jquery.slim.min.js

- | | | └─ jquery.slim.min.map
- | | └─ jquery-easing
- | | └─ jquery.easing.compatibility.js
- | | └─ jquery.easing.js
- | | └─ jquery.easing.min.js
- | └─ urls.py
- | └─ wsgi.py
- └─ DDRA
 - | └─ DR_classes.txt
 - | └─ __init__.py
 - | └─ __pycache__
 - | | └─ __init__.cpython-36.pyc
 - | | └─ admin.cpython-36.pyc
 - | | └─ engine.cpython-36.pyc
 - | | └─ forms.cpython-36.pyc
 - | | └─ models.cpython-36.pyc
 - | | └─ urls.cpython-36.pyc
 - | | └─ views.cpython-36.pyc
 - | └─ admin.py
 - | └─ apps.py
 - | └─ engine.py
 - | └─ forms.py
 - | └─ models.py
 - | └─ tests.py
 - | └─ urls.py
 - | └─ views.py
- └─ componentDiagram.txt
- └─ datasetsImages

```
| |— datasetsImages
| | |— example.jpeg
| | |— graffiti2.jpg
| | |— unknow.jpeg
| |— example.jpeg
|— db.sqlite3
|— manage.py
|— models
| |— DR_classes.txt
| |— export.pkl
|— requirment.txt
|— templates
|   |— base.html
|   |— display.html
|   |— home.html
```

19 directories, 102 files





















(b) Deployment Diagram (created using draw.io):

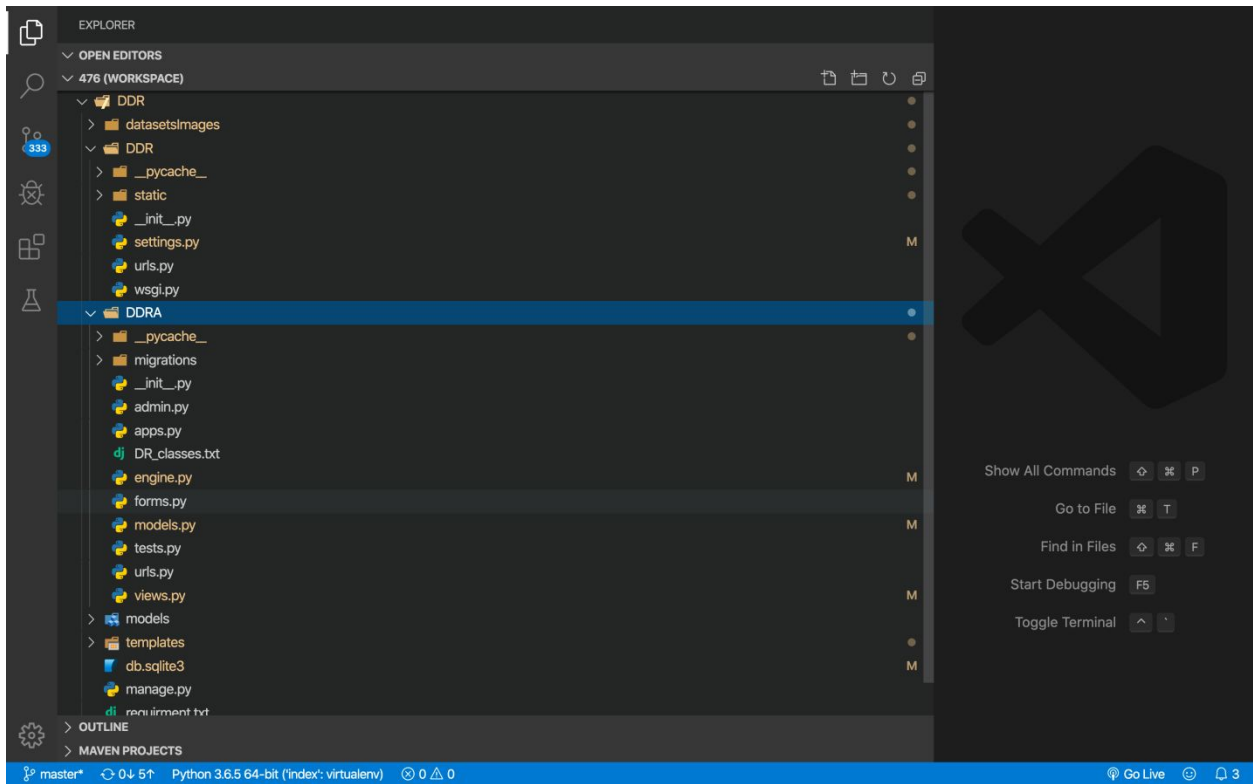


(c) Screenshots of table of contents of system data:

OPEN FILES

FOLDERS

- ▼  Venv ●
 - ▶  bin
- ▼  DDR ●
 - ▶  datasetsImages ●
 - ▼  DDR ●
 - ▶  __pycache__ ●
 - ▼  static ●
 - ▶  css ●
 - ▶  js
 - ▶  vendor
 - /* gulpfile.js
 - /* __init__.py
 - /* settings.py ●
 - /* urls.py
 - /* wsgi.py
 - ▼  DDRA ●
 - ▶  __pycache__ ●
 - ▶  migrations
 - /* __init__.py
 - /* admin.py
 - /* apps.py
 - ≡ DR_classes.txt
 - /* engine.py ●
 - /* forms.py
 - /* models.py ●
 - /* tests.py
 - /* urls.py
 - /* views.py ●
 - ▶  models
 - ▼  templates ●
 - <> base.html ●
 - <> display.html ●
 - <> home.html ●
 -  db.sqlite3 ●
 - /* manage.py
 - ≡ requirment.txt
 - ▶  include
 - ▶  lib ○
 -  .json
 -  .Python



6. Technical Documentation

(a) The following languages were used for this project:

- Python
 - **Neither of us had any experience with Python prior to beginning this project.**
 - However, because Python is one of the languages of choice for artificial intelligence, it was essential that we begin learning to use it.
 - As well, many of the preexisting projects found on the Internet were dependent upon Python. In order to attempt to understand them and possibly make use of them, we needed to first get acquainted with the language.
- JavaScript
 - No explanation needed; this is one of the gold standards of client-side functionality.

- HTML
 - Used for front-end design and functionality, alongside JavaScript.

(b) The following algorithms and/or programs were used for this project:

- The Kaggle user “*tanlikesmath*” has provided a well-documented notebook as part of their Kaggle competition entry:
 - <https://www.kaggle.com/tanlikesmath/diabetic-retinopathy-with-resnet50-oversampling>
- The GitHub user “*tmabraham*” has provided a repository that employs a SqueezeNet and the above-linked Kaggle ResNet50 solution.
 - <https://github.com/tmabraham/retinopathy-classifier>

(c) The following software tools and environments were used for this project:



- **Django:**
 - **Neither of us have worked with Django before.** It is a free and open source web framework written in Python. It allowed us to use the unique MVT architecture mentioned above in Section 4.
 - Although we were not familiar with using it, we decided it would be an ideal tool for this project because it uses Python to deliver solutions across a well-organized web architecture.

- As well, the bulk of the code that we looked at was already written in Python (e.g. the neural networks demonstrated on Kaggle are mainly written in Python). By choosing Django, we were able to reduce the number of languages needed and keep Python involved in both the front and back-ends.
- **VS Code (Visual Studio Code for Python):**
 - This was our primary IDE for developing code. VS Code was a very useful tool in this project because it is versatile enough to handle multiple languages (e.g. Python, JavaScript).
- **jQuery:**
 - jQuery is a well-known JavaScript library which can be used to build user interface (UI) components. Its front-end capabilities make it a good match for Django.
- **Jupyter Notebook:**
 - The Jupyter Notebook is an app for writing code as well as documenting, via text and imagery, what the project is meant to demonstrate. This makes it a great teaching tool, and many of the Kaggle competitors demonstrated their projects via Jupyter Notebooks.
 - More importantly, you can easily run live code in a browser if you want to test someone else's code. This was especially important during the early stages of our project when we were unsure of any functionality and could test it out online without having to install anything locally.
 - **This was our first time experimenting with Jupyter Notebook.**
- **LucidChart:**
 - LucidChart is an excellent web-based software tool for creating charts and diagrams.
 - Because this is a team project, the ability of LucidChart to share documents between group members allowed for enhanced cooperation and communication.
- **Draw.io:**

- Draw.io is a free web-based UML design tool, similar to LucidChart, and easy to use without logging in.
- **Trello:**
 - Trello is an organization and productivity tool. Although not directly integrated in the project itself, it was extremely useful during the design phase.
 - Specifically, it allowed us to delegate tasks among group members and break tasks into sub-tasks (including the sections of this report) so that we could more easily track our overall progress.
- **Bootstrap Front-End Theme:**
 - We used a simple, free theme for the website's interace.
 - In keeping with the commonly accepted heuristics for usability, we opted not to use an unnecessarily cluttered design, choosing a minimalistic design.



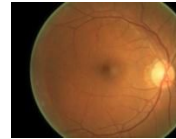

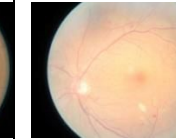
7. Acceptance Testing

(a) Functional Testing

We now demonstrate five examples of functional testing using drastically different input images. Our main goal is to ascertain the correctness of output:

- Does the image upload successfully?
- If so, does the software generate an accurate diagnosis (i.e. in agreement with a professional healthcare provider)?

Here is a preview of the input images we will be providing:

	Patient 1	Patient 2	Patient 3	Patient 4	Patient 5
Retinal Imaging					
Diagnosis given by clinician: (expected output)	Healthy. No DR detected.	Diseased; only a mild amount of DR was detected (grade I).	Diseased; moderate amount of DR was detected (grade II).	Diseased; severe amount of DR was detected (grade III).	Diseased; proliferative retinopathy was detected (grade IV).

We need to confirm whether or not our software agrees with the clinician grades, which came from a human expert. Let's start testing.

□ *Note that the input screenshots have the name of the input visible in the upload field. For example, in Input screenshot 1, you can see the input labelled as “No_Retinopathy.jpg”*

Patient 1:

Input:

Detecting Diabetic Retinopathy

Home About Github

No_Retinopathy.jpg

Browse

Diabetic Retinopathy

Diabetic retinopathy (DR), also known as diabetic eye disease, is a medical condition in which damage occurs to the retina due to diabetes mellitus. It is a leading cause of blindness. Diabetic retinopathy affects up to 80 percent of those who have had diabetes for 20 years or more. Diabetic retinopathy often has no early warning signs. Retinal (fundus) photography with manual interpretation is a widely accepted screening tool for diabetic retinopathy, with performance that can exceed that of in-person dilated eye examinations.

This website detects Referrable Diabetic Retinopathy (RDR) using Deep Learning and was built with fastai.

Output:

Detecting Diabetic Retinopathy

Home Result Github

Upload Another File

Browse

Results :

No Referrable Retinopathy : 0.85

Referrable Diabetic Retinopathy Present : 0.15

Image :



The software has diagnosed this patient as healthy with a probability of 85%.



This is in agreement with the human expert, who gave a similar diagnosis.

Patient 2:

Input:

Detecting Diabetic Retinopathy

Home About Github

mild_retinopathy.jpg Browse

Diabetic Retinopathy

Diabetic retinopathy (DR), also known as diabetic eye disease, is a medical condition in which damage occurs to the retina due to diabetes mellitus. It is a leading cause of blindness. Diabetic retinopathy affects up to 80 percent of those who have had diabetes for 20 years or more. Diabetic retinopathy often has no early warning signs. Retinal (fundus) photography with manual interpretation is a widely accepted screening tool for diabetic retinopathy, with performance that can exceed that of in-person dilated eye examinations.

This website detects Referrable Diabetic Retinopathy (RDR) using Deep Learning and was built with fastai.

Output:

Detecting Diabetic Ratinopathy

HomeResultGithub

Upload Another File

Browse

Results :

No Referrable Retinopathy : 0.64

Referrable Diabetic Retinopathy Present : 0.36

Image :



The software has diagnosed this patient with mild / borderline diabetic retinopathy
with a probability of 0.36 (36%).



This is in agreement with the human expert, who gave a similar diagnosis.

Patient 3:

Input:

Detecting Diabetic Ratinopathy

Home About Github

moderate.jpg

Browse

Diabetic Retinopathy

TDiabetic retinopathy (DR), also known as diabetic eye disease, is a medical condition in which damage occurs to the retina due to diabetes mellitus. It is a leading cause of blindness. Diabetic retinopathy affects up to 80 percent of those who have had diabetes for 20 years or more. Diabetic retinopathy often has no early warning signs. Retinal (fundus) photography with manual interpretation is a widely accepted screening tool for diabetic retinopathy, with performance that can exceed that of in-person dilated eye examinations.

This website detects Referrable Diabetic Retinopathy (RDR) using Deep Learning and was built with fastai.

Output:

Detecting Diabetic Ratinopathy

Home Result Github

Upload Another File

Browse

Results :

Referrable Diabetic Retinopathy Present : 0.65

No Referrable Retinopathy : 0.35

Image :



The software has diagnosed this patient with diabetic retinopathy with a probability of 0.65 (65%).



This is in agreement with the human expert, who gave a similar diagnosis.

Patient 4:

Input:

Detecting Diabetic Retinopathy

Home About Github

severe.jpg

Browse

Diabetic Retinopathy

Diabetic retinopathy (DR), also known as diabetic eye disease, is a medical condition in which damage occurs to the retina due to diabetes mellitus. It is a leading cause of blindness. Diabetic retinopathy affects up to 80 percent of those who have had diabetes for 20 years or more. Diabetic retinopathy often has no early warning signs. Retinal (fundus) photography with manual interpretation is a widely accepted screening tool for diabetic retinopathy, with performance that can exceed that of in-person dilated eye examinations.

This website detects Referrable Diabetic Retinopathy (RDR) using Deep Learning and was built with fastai.

Output:

Detecting Diabetic Retinopathy

Home Result Github

Upload Another File

Browse

Results :

Referrable Diabetic Retinopathy Present : 0.95

No Referrable Retinopathy : 0.05

Image :



The software has diagnosed this patient with diabetic retinopathy with a probability of 0.95 (95%).



This is in agreement with the human expert, who gave a similar diagnosis.

Patient 5:

Input:

Detecting Diabetic Retinopathy

Home About Github

last.jpg

Browse

Diabetic Retinopathy

Diabetic retinopathy (DR), also known as diabetic eye disease, is a medical condition in which damage occurs to the retina due to diabetes mellitus. It is a leading cause of blindness. Diabetic retinopathy affects up to 80 percent of those who have had diabetes for 20 years or more. Diabetic retinopathy often has no early warning signs. Retinal (fundus) photography with manual interpretation is a widely accepted screening tool for diabetic retinopathy, with performance that can exceed that of in-person dilated eye examinations.

This website detects Referrable Diabetic Retinopathy (RDR) using Deep Learning and was built with fastai.

Output:

Detecting Diabetic Retinopathy

Home Result Github

Upload Another File

Browse

Results :

Referrable Diabetic Retinopathy Present : 0.96

No Referrable Retinopathy : 0.04

Image :



The software has diagnosed this patient with diabetic retinopathy with a probability of 0.96 (96%).

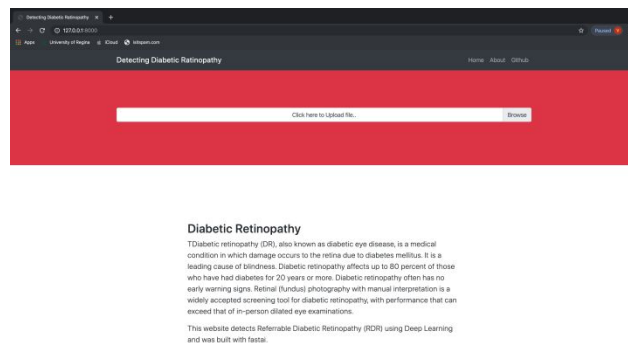
✓ This is in agreement with the human expert, who gave a similar diagnosis.

(b) Robustness Testing

We investigated five cases where robustness was necessary:

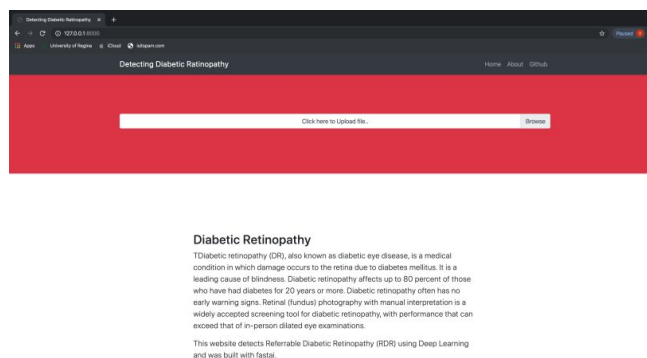
Case 1: User attempts to submit an empty field (no file).

Solution: Software remains on home page and is unable to upload unless a file is in place. An empty field has no way of being submitted.



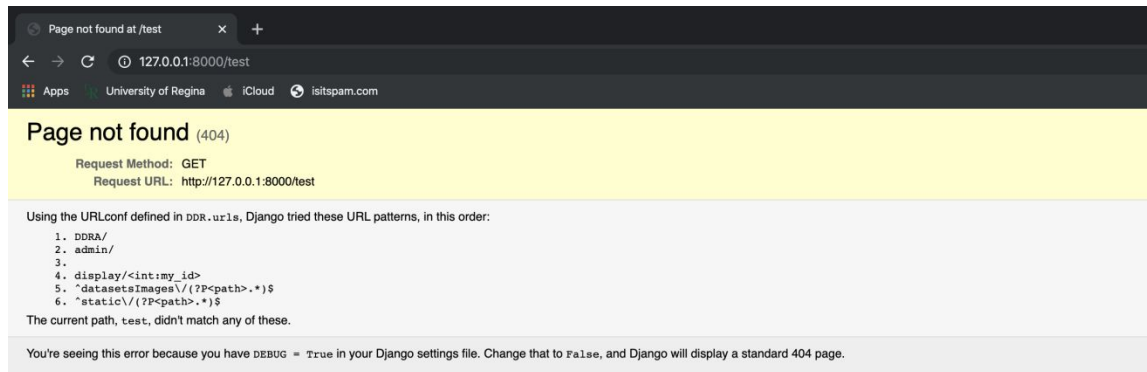
Case 2: User attempts to upload a file but it is not of image type (e.g. user accidentally selects a PDF).

Solution: Software detects that an inappropriate file type was attempted to be uploaded. No file is uploaded and the user remains on the home page.



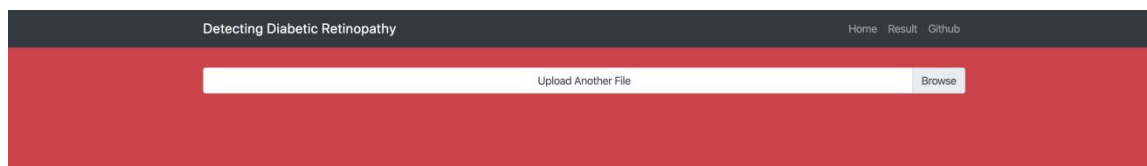
Case 3: User attempts to access an invalid URL intentionally or unintentionally by altering the address in their browser (e.g. to access another page within the site or a URL which is internal to Django).

Solution: The user is redirected to an appropriate error message (404).



Case 4: User uploads an image, but it is not a proper funduscopy or ophthalmoscopic image (e.g. an image of a car).

Solution: The software will accept the upload of the image because it is a valid file type, but it will not detect any diabetic retinopathy as there is none present.



Results :

Referrable Diabetic Retinopathy Present : 0

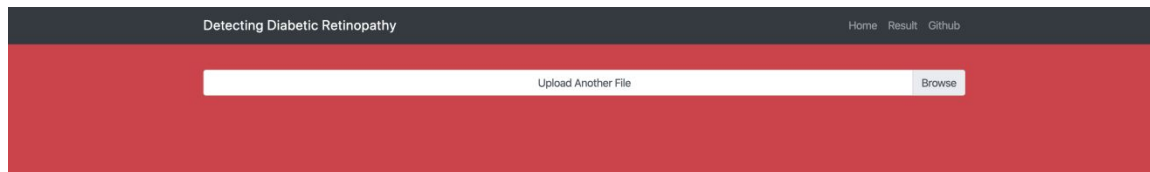
No Referrable Retinopathy : 100

Image :



Case 5: User uploads an image of an eye which has been overlaid with graphical / textual data (e.g. the image is partially obscured)

Solution: The software is still able to detect retinopathy in a diseased patient. However, its accuracy is dependent on the amount of noise or pollution present in the image. If too much of the image is covered up, we essentially return to Case 4.

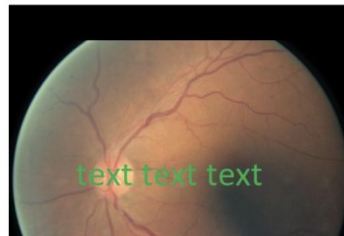


Results :

Referrable Diabetic Retinopathy Present : 0.75

No Referrable Retinopathy : 0.25

Image :



(We placed graffiti (“text text text”) over an image to see whether disease was still detected successfully.)

(c) Time-Efficiency Testing

We will be using the following rubric to evaluate the performance of our local app versus a preexisting web-based app created by **tmabraham** on GitHub (linked in the technical documentation as well as the feasibility study).

Evaluation rubric:

- Less than 1 sec : **Excellent performance**
- Between 1 and 3 sec : **Average / mediocre**
- More than 3 sec : **Poor**

Our project will be referred to below as DDR (Detection of Diabetic Retinopathy).

We will be comparing our project against the Heroku app mentioned above; this is a pre-existing project and not ours. It is a web implementation of the GitHub user tmabraham's code, which we have linked multiple times throughout this report, as this program serves as a good challenge / comparison to ours.

Case 1 – Loading the website:

Heroku: *More than 3 sec; poor performance.* We believe that Heroku apps go into standby mode and are not woken until someone attempts to access them. This causes a lengthy wake-up wait-time. In this case, the Heroku app took 41 seconds just to start up.

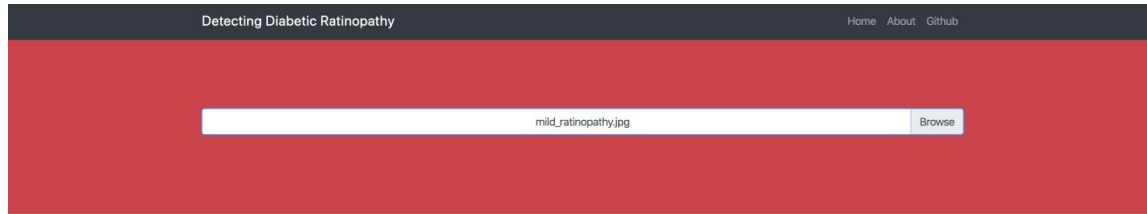


Screenshot: *Our app was competing against Heroku (above), which took 41 seconds to load the first time.*

DDR (our project): Less than 1 sec; excellent load time.

Case 2 – Uploading an image:

Heroku: *Between 1 and 3 seconds; average / mediocre performance.* Note that the app design we are comparing against requires more manual input and clicking from users, so uploading an image takes longer. Our app, below, does not have this issue:



Diabetic Retinopathy

Diabetic retinopathy (DR), also known as diabetic eye disease, is a medical condition in which damage occurs to the retina due to diabetes mellitus. It is a leading cause of blindness. Diabetic retinopathy affects up to 80 percent of those who have had diabetes for 20 years or more. Diabetic retinopathy often has no early warning signs. Retinal (fundus) photography with manual interpretation is a widely accepted screening tool for diabetic retinopathy, with performance that can exceed that of in-person dilated eye examinations.

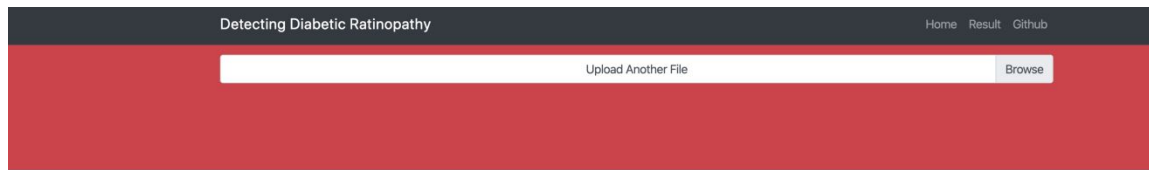
This website detects Referrable Diabetic Retinopathy (RDR) using Deep Learning and was built with fastai.

DDR (our project): Less than 1 sec; excellent performance. Our app immediately uploads the selected file without requiring any further clicking. This saves time, and the automation also makes the appx more productive and user friendly.

Case 3 – Re-uploading an image (or choosing subsequent new images):

Heroku - *More than 3 sec : poor performance.* This is due in large part to the fact that the Heroku app requires the user to input multiple clicks and/or button presses for every image.

DDR - Less than 1 sec : excellent performance. Our app is extremely efficient when it comes to uploading images in quick succession. While viewing the results of one image you can immediately upload another. There is no sequence of buttons to press. As you can see from the screenshot, the app is immediately ready to have another file uploaded:

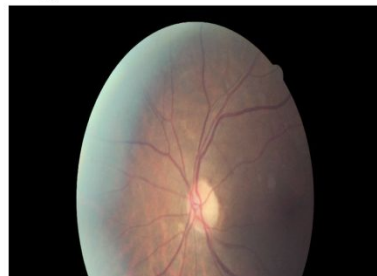


Results :

No Referrable Retinopathy : 0.85

Referrable Diabetic Retinopathy : 0.15

Image :



Case 4 – Time to receive diagnosis:

Heroku - More than 3 sec : poor. This may result from our competitor's use of networked threading. It may also rely on Internet connection speeds and image size. For fairness, we used identical images.

DDR - Less than 1 sec : excellent turn-around time.

Case 5 – Routing between different pages of the website:

Heroku - Average / mediocre navigation of 1 – 3 seconds. Due to Heroku's somewhat slow speed, it takes a while to navigate between different pages.

DDR - Less than 1 sec : excellent performance. It is worth noting that different pages exist within our app, but the layout is kept consistent so that the user feels like only some content has changed.

[End of report.]